Karlstad Universitet

Högskoleingenjörsprogrammet i Elektroteknik

Vision based control and landing of Micro aerial vehicles

Bachelor Thesis
Department of Engineering and Physics
May 27, 2019

Author:
Christoffer Karlsson

Course:
Degree project (ELGC11)

Internal supervisor:
Jorge Solis

External supervisor:
Kristoffer Richardsson

Acknowledgements

I would hereby like to express by my greatest appreciation to all the people who helped and made it possible for me to perform this work.

Firstly, I would like to thank Assoc. Prof Jorge Solis for the continuous support throughout this thesis. His guidance incented me to widen my research from various perspectives and he gave me insightful input throughout the project. It is also thanks to him that I came in contact with Bitcraze and I would not have been able to do this particular project if it were not for him.

Secondly, I would like to thank Kristoffer Rickardsson at Bitcraze AB for all the support, encouragement and inspiration during the project.

I would also like to give a sincere thanks to all my classmates for the stimulating discussions and all the fun we have had during the years at Karlstad University.

I am also immensely grateful to my family for their unconditional support and encouragement.

Finally, I want to express my profound gratitude to my beloved fiancée, Sofie Dervander, who has supported me throughout my years of study, carrying me through stressful times and were always there for me when I needed.

Abstract

This bachelors thesis presents a vision based control system for the quadrotor aerial vehicle, Crazyflie 2.0, developed by Bitcraze AB. The main goal of this thesis is to design and implement an off-board control system based on visual input, in order control the position and orientation of the vehicle with respect to a single fiducial marker. By integrating a camera and wireless video transmitter onto the MAV platform, we are able to achieve autonomous navigation and landing in relatively close proximity to the dedicated target location.

The control system was developed in the programming language Python and all processing of the vision-data take place on an off-board computer. This thesis describes the methods used for developing and implementing the control system and a number of experiments have been carried out in order to determine the performance of the overall vision control system.

Sammanfattning

I detta examensarbete presenteras ett visionsbaserat kontrollsystem för drönaren Crazyflie 2.0 som har utvecklats av Bitcraze AB. Målet med detta arbete är att utforma och implementera ett externt kontrollsystem baserat på data som inhämtas av en kamera för att reglera fordonets position och riktning med avseende på en markör placerad i synfältet av kameran. Genom att integrera kameran tillsammans med en trådlös videosändare på plattformen, visar vi i denna avhandling att det är möjligt att åstadkomma autonom navigering och landning i närheten av markören.

Kontrollsystemet utvecklades i programmeringsspråket Python och all processering av visions-datan sker på en extern dator. Metoderna som används för att utveckla kontrollsystemet och som beskrivs i denna rapport har testats under ett flertal experiment som visar på hur väl systemet kan detektera markören och hur väl de olika ingående komponenterna samspelar för att kunna utföra den autonoma styrningen.

Contents

Chapter 1

Introduction

1.1 Introduction

In recent years, the interest around micro aerial vehicles (MAVs) has grown rapidly, both for use within the industry as well as for hobbyists and for research purposes. The small size of the craft allows for applications such as remote observation or analysis of environments that would be otherwise inaccessible to a larger vehicle or individuals. The small size of the MAV also facilitate indoor flight in tightly constrained environments or in close proximity to people without significantly exhibiting a hazard, as well as promote the overall agility of the vehicle.

Some of the reasons behind the growing popularity around quadrotor MAVs, also known as quadcopters, include the relatively simple mechanical structure of the vehicle, as well as the high manoeuvrability and low price, allowing for a broad user base and high accessibility. Another reason behind the growing interest around quadrotors is their ability to handle various payloads within reasonable limits, which allows for modifications to be made in order for the vehicle to perform various tasks. Recent technological innovations in micro-controllers and sensors has paved the way for research around autonomous flight which will also be the context of this thesis.

MAVs can be classified into three different types, fixed wing configuration, rotary wing configuration and flapping wing configuration, each with their own advantages and disadvantages [20]. This thesis will employ the *Crazyflie 2.0*, a low weight rotary wing quadrotor from the Swedish company, Bitcraze AB, released as an open source development platform, to which we mount a RGB CMOS camera and a wireless analogue video transmitter.

One of the major complications around quadrotors are that they are inherently unstable without the assistance of a control system. They must rely on sensor data and constant corrections to be made at a very high rate to maintain stability. If the on board electronic control unit (ECU) would fail for only a short amount of time, it could bring the quadrotor into a crash and thereby constitute a danger to both the vehicle itself, the environment or human bystanders. Since MAVs often rely on a lithium battery as the primary source of power, as the size of the vehicle is decreased, the size of the battery must also be dimensioned appropriately in order to achieve a feasible thrust to weight ratio. By limiting the size of the battery, the accessible amount of energy available to the MAV becomes as a result reduced as well. If the battery level is running low whilst the craft is in flight, it must be able to make a good estimation of the remaining time it can stay in flight before it needs to land and recharge the batteries, or it would simply crash as a result of depleted batteries. In the case of low battery level under circumstances where the vehicle is human controlled, the pilot would need to locate a safe zone for landing. However, an autonomous MAV must be able to find a safe location to land and recharge the batteries on its own, without any human interaction. This can be an especially challenging scenario in GPS denied areas or when GPS is not an available option.

With this presupposition in mind, I will in this thesis present a method for vision based control and landing of a quadrotor utilizing a small RGB-camera, analogue video transmitter and a pre-defined marker depicting the landing zone.

1.1.1 Related work

In past years, several papers have been published regarding research around autonomous flight for quadrotors using one or multiple cameras. The major difference among them is the implementation of the control algorithm and the selection of the fiducial markers. Some of them rely on systems where the trajectory computations are done off-board and others have implemented solutions where the vision-data is processed on-board. In the case of this project however, on-board computations are not a viable option because of the limited computational power available on the MAV platform, but may be possible for future generations of the Crazyflie platform. For instance, [10] presents a quadrotor system capable of achieving autonomous navigation and landing on a moving platform using only on-board sensing and computing. A similar concept using optical flow and vision based detection is presented by [23], where the marker depicting the target landing location consists of several concentric white rings on a black background. This project also rely on on-board computation where the embedded computer receives images from an on-board camera.

Furthermore, [11] has demonstrated a method for achieving autonomous hovering using a monocular SLAM-system for the Crazyflie. With this method, the Crazyflie is able to remain stable whilst in flight without the need of using any external sensors or relying on the optical flow deck which is used in this project. However, [11] relies on pose estimation through the Kinect RGB-D sensor, which in contrast to the method described in this thesis, must be configured as an external set-up, meaning that the control system requires data from sensors external to the MAV platform in order to achieve autonomous waypoint flying. Various research has also been done on methods for controlling the position of a quadcopter by using control systems other than the PID controller. For example, [28] proposes a method where an LQ-controller is used together with a PID controller in order to control the position of the quadcopter in all three dimensions.

1.1.2 Problem statement

The goal of this thesis is to develop a method for achieving autonomous flight based on fiducial markers for a quadrotor MAV, primarily concerning the following points:

- Navigation and landing should be performed without any human interaction.
- The solution should not be dependant on sensor information external to the MAV.
- The visual control system should be able to detect a defined fiducial marker in the environment around the MAV and verify the validity of the marker.
- The system should be able to infer information from a single defined marker in its environment, allowing the MAV to localize itself relative to the target location.
- The effective flight time should not be adversely affected by more than 50 percent due to additional sensors and peripherals.
- The MAV should be able to hover, navigate and fly to the desired target location while remaining stable.

1.1.3 Proposed solution

In order to satisfy the conditions mentioned above in section 1.1.2, some considerations has to be made. The camera and video transmission module should be as light weight as possible while still maintaining good resolution and image quality. I opted for using a small commercial off-the-shelf video transmitter and camera module from Eachine, carrying a total weight of 4 grams[7]. The Crazyflie 2.0 from Bitcraze has a specified maximum payload capacity of 15 grams[2], making the camera- and transmitter module fit well within the specified payload range. The transmitter and camera module is mounted on the top side of the quadcopter and the video feed containing visual information about the environment around the MAV is transmitted through wireless communication to the PC which handles the computations of the control signals that are to be sent back to the quadcopter.

I will in this thesis evaluate two different methods for visual odometry, both extraction of image feature points, known as the *Feature Based Method*, and the *Direct Method*, where the pixel intensities are used as the direct visual input. Estimation of the camera motion will be done by utilizing a homography transformation and a linear Kalman filter which will also aid in reducing the noise in the discrete data signals. In order to minimize drift of the quadcopter in the inertial frame, I will make use of the *Flow deck v2* from Bitcraze, which give the quadcopter the ability to dicern movement above the ground.

The fiducial marker specifying the target landing location must be clearly identifiable and verifiable from the camera perspective, thus it should be in a shape and form that stands out from the rest of the environment, as well as provide some form of error detection. I decided to use *ArUco*, an open source marker dictionary and library that also offer camera pose estimations using square planar markers [31][13], but I will also evaluate the feature matching algorithm *ORB (Oriented FAST and Rotated BRIEF)*, which is an open source alternative to *SIFT* or *SURF*[9] which are both feature detection and descriptors used for e.g. object recognition and classification.

1.1.4 Delimitations

Research around autonomous rotary wing quadrotors include a broad spectrum of objectives, questions and variables, thus it is necessary to confine the work within some bounds. My interest in this subject lies primarily within the process of control using visual odometry, hence the delimitations are set up as accordingly and with consideration to the time limit of the project:

- Throughout the whole project, testing, analysis and verification will be conducted for indoors conditions, i.e., contributing factors such as wind and ambient temperature will be kept near constant.
- The project will not include the process of charging the battery once the MAV has landed.
- I will use a non-model based approach for tuning the control system parameters
- Because of the limited size of the workspace where I do my experiments, the small form factor of the camera and its relatively low resolution, the detection of the fiducial marker should be limited to a maximum of two meters between the camera lens and the marker.
- I will not consider cases with various payloads other than the flow deck, camera and transmission module.
- The focus on this thesis will be on controlling the MAV through visual input data by communication with a ground station computer and therefore the control parameters internal to the firmware of the MAV will be left unadjusted if not explicitly required.

1.1.5 Outline

- In chapter 1, we have summarized the intentions of this thesis and defined the problem statement. We have proposed a solution and defined the delimitation of which we confine the work. We will also cover some basic notations, conventions and terminology that will be used throughout this thesis.
- In chapter 2 we take a look on the hardware platform on which the vision system is implemented in order for the reader to get a better understanding of the upcoming chapters. The chapter will cover some basics about the vehicle itself, as well as the hardware used for data acquisition and communication between the MAV and the ground-station computer.
- In chapter 3 we discuss the underlying theory behind the concepts implemented in the vision control system, the basic operation of the camera and we introduce the reader to the *ArUco library* used for detection and pose estimation.
- In chapter 4, we examine how the hardware- and software components in the system were implemented to work together and take a closer look on the general control structure and the algorithm for achieving autonomous navigation and landing towards the target.
- In chapter 5 we cover the outcome of the experiments conducted during and after development of the vision based control system and evaluate the performance of the control system and the detection algorithms, as well as investigate how the flight time is affected by adding additional sensors.
- Finally, in chapter 6, we will conclude the results of this project and discuss future work.

1.2 Conventions and terminology

This section will cover the some of the most important conventions, notations and terminology used throughout this thesis.

Table 1.1: Notations used throughout the thesis

Notation description	Example
Matrices are denoted in bold upper-case letters	$\boldsymbol{H}$
Identity matrix	$\boldsymbol{I}$
Vectors are denoted either as bold lower-case letters or with an arrow	$\boldsymbol{x}$, $\overrightarrow{\boldsymbol{X}}$
Scalar values are denoted as italic lower-case letters	x
Three dimensional Euclidean space	$\mathbb{R}^3$
Planar two-dimensional space	$\mathbb{P}^2$
Image frame coordinates are represented by $u_{x,y}$	(u_x, u_y)
Reference frames are denoted in calligraphic capitals	$\mathcal{M}$
Leading superscript denote the frame of which the object occupy	${}^{\mathcal{C}}\mathbf{P}$
An object in frame $\mathcal{M}$ transformed to frame $\mathcal{C}$	${}^{C}\mathbf{P}_{\mathcal{M}}$
The camera reference frame	$\mathcal{C}$
The marker reference frame	$\mathcal{M}$
Time derivatives are indicated by a dot or by *Leibniz's notation*	$\dot{\boldsymbol{x}}$, $\frac{d}{dt}\boldsymbol{x}$

Covariance matrix:

Each element (i, j) in the covariance matrix describe the degree of correlation between the $i : th$ state variable and the $j : th$ state variable. We define a covariance matrix as:

$$\boldsymbol{\Sigma} = \begin{bmatrix} \Sigma_{ii} & \Sigma_{ij} \\ \Sigma_{ji} & \Sigma_{jj} \end{bmatrix} \tag{1.1}$$

Standard deviation

The standard deviation is a statistical measurement that quantify the amount of variation in a set of discrete values and is in this thesis denoted by σ, which we define as:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2} \tag{1.2}$$

, where N is the size of the data set, μ is the mean value and x_i denote an individual element in the data set.

3D rotation group

The group of all rotations about the origin in three-dimensional Euclidian space $\mathbb{R}^3$:

$$SO(3) := \boldsymbol{R} \in \mathbb{R}^3 | \boldsymbol{R}^T\boldsymbol{R} = \boldsymbol{I}, det(\boldsymbol{R}) = \pm 1 \tag{1.3}$$

Framestore:

The camera captures its source information to a framestore in the computer which is a portion of memory containing a spatial sample of the image. The pixels in the framestore can be seen as a vector function (bitmap or image function) whose components represent the values of the physical quantities captured by the camera in a sampled and quantized form.

Grayscale image:

A grayscale image is represented in the computer as a $m \times n$ matrix where the elements contained in the matrix: $\boldsymbol{J}_{ij} \in [0, 255]$, where $\boldsymbol{J}$ is the image function and 0 represent a completely black pixel and 255 represent a completely white pixel. The elements of $\boldsymbol{J}$ are defined as integers.

Color image:

Color images following the *RGB color model* are represented as three grayscale image matrices, one for each color channel. The image function for such images are:

$$\boldsymbol{J} = \boldsymbol{J}_R(x_J, y_J), \boldsymbol{J}_G(x_J, y_J), \boldsymbol{J}_B(x_J, y_J) \tag{1.4}$$

, where each component correspond to the light intensity in its respective color channel. The number of gray-levels (0-255) depend of the adapted grayscale resolution but is always bound to the minimum and maximum light intensity in physical quantities.

OpenCV

OpenCV is an open source library built natively in the C++ programming language to provide algorithms for applications in computer vision and machine learning software. The algorithms contained in the OpenCV library can be used to e.g., detect and identify objects, track camera movement and camera calibration and is has cross-platform support for Python, C++, Java and MATLAB.

Thresholding

In image processing, thresholding is a method of image segmentation and can be used to form binary images from a grayscale- or color image by replacing a pixel if the image intensity in $\boldsymbol{J}_{i,j}$ is less than or greater than a defined scalar value. In this thesis, *Otsu's method* is used for thresholding an image and the interested reader is referred to [19] for more information about this topic.

Chapter 2

Hardware

In order to get an idea about the system configuration, we will in this section cover all the hardware used during this project. We will begin with presenting the the platform and describe its components and features and then take a closer look at the peripherals and sensors used for communication and data collection.

2.1 Crazyflie 2.0 Quadcopter

The word *quadcopter* is derived from *"quadrotor helicopter"*, which imply that they exhibit flight characteristics similar to helicopters, but with the thrust generated from a four-rotor configuration. The quadcopter has similar vertical take-off and landing (VTOL) capabilities of a helicopter and the ability to hover while remaining perfectly stable, combined with excellent manoeuvrability. Modern quadrotor aircrafts come in many forms and sizes, from very small designs fitting in the palm of the hand, to very large configurations, targeted towards use in e.g., the military, agricultural purposes or for use within the film- and photography industry.

The commercially available Crazyflie 2.0 from Bitcraze AB is used as platform for this project. It is an extremely lightweight quadrotor MAV, designed with simplicity and development in mind, allowing for software and hardware modifications to be made with relative ease. Bitcraze AB was founded in 2011 with the purpose of financing and developing the Crazyflie-kit as an open source development platform [1] and shipped their first units in 2013. Thanks to the success of their initial product, they started the development of the Crazyflie 2.0 in 2014 with the ambition to create a versatile platform with the possibility of hardware expansion and customization.

2.1.1 Specifications

The Crazyflie 2.0 comes equipped with a dual main control unit (MCU) architecture with dedicated power- and radio management. The nRF51822 is an always powered on slave unit that handles the radio communication with the computer and the power management. It also communicate the data via UART to the STM32F405 (see figure 2.1) and detects and check installed expansion boards. The STM32F405 gets initiated by the nRF51822 and acts as a master which runs the main firmware of the whole system.

In its base configuration, the MAV has a takeoff weight of only 27 grams and a size of 92x92x29 millimetres[2]. As stated on the Bitcraze website, the flight time of the Crazyflie 2.0 with mounted stock battery is 7 minutes, which can in consonance with experiments covered in this theses, considered quite to be accurate.

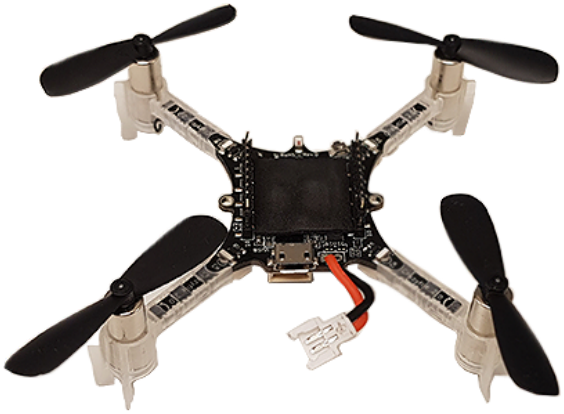


Figure 2.1: With a specified take-off weight of only 27 grams, the Crazyflie 2.0 from Bitcraze AB makes an excellent indoors flying platform

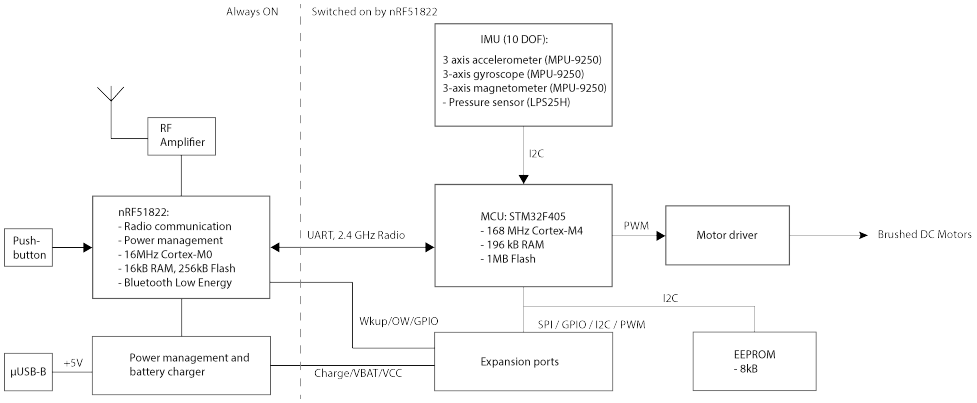


Figure 2.2: System architecture of the Crazyflie 2.0

To keep the quadrotor level autonomously, some additional proprioceptive sensors are required, which in the case of this platform comes packaged as an inertial measurement unit (IMU) with 9 degrees of freedom(9DOF). The quadrotor must have a notion of its relative position and velocity, as well as the rate of change of velocity and its orientation to the earth in order to determine its own movement along any of its axis. The Crazyflie 2.0 comes with the MPU-9250 from Invensense, which is a micro-electro-mechanical systems (MEMS) based IMU with an accelerometer which measures the inertial force generated when the quadrotor is affected by a change in velocity, a gyroscope that measures the rate of rotation and a digital compass which measures the strength and direction of local magnetic fields. The Crazyflie 2.0 also comes with a high precision absolute air pressure sensor, the LPS25H, which can be used for approximating the altitude of the craft above sea level.

The battery used is a Lithium-Polymer (LiPo) battery with a nominal voltage of 3.7V and a rated capacity of 250mAh, giving it a theoretical capacity of 925mWh. It also comes with a Protection Circuit Module (PCM) attached, which prevents the user from under- or over charging the battery, as well providing short circuit protection. The battery is easily swappable thanks to its JST-DS connector.

The Crazyflie 2.0 uses four 7x16 millimetres brushed DC motors, utilizing one pull-down MOSFET-transistor per motor, controlled by a 328kHz PWM-signal. Thus, the motor driver is designed such that each rotor should only spin in one particular direction, either clockwise

or counter-clockwise. The four rotors are set up in pairs of two with a common angular direction. One of the pairs rotate in clockwise direction, whilst the other pair rotate in the opposite direction, see figure 2.3 below.

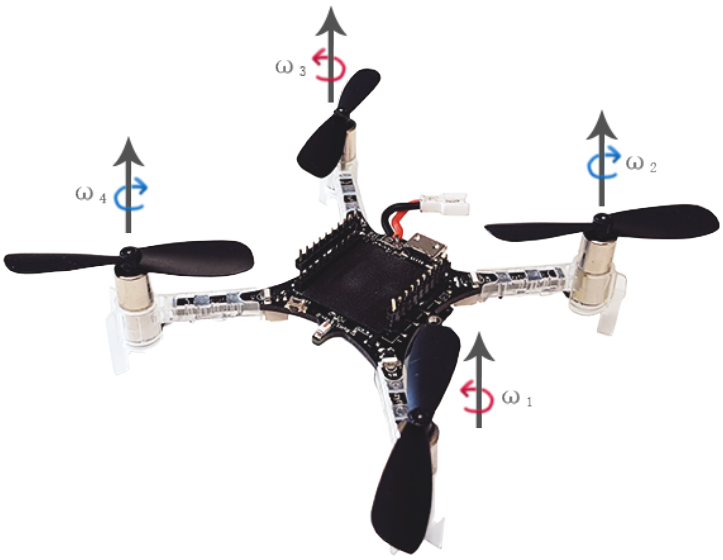


Figure 2.3: One motor pair $(2, 4)$ turns clockwise and the other pair $(1, 3)$ turns counter-clockwise. By varying the absolute value of the angular velocity, $\boldsymbol{\omega}_i$ of any of the four motors, the position and orientation of the craft may be controlled.

Communication between the Crazyflie 2.0 and a client computer is achieved with the *Crazyradio PA 2.4GHz USB-dongle*, based on the nRF24LU1+ from Nordic Semiconductor [3]. The Crazyradio PA allows for wireless two-way communication via USB with up to 2Mbit/s data-rate, making it quick and easy to both send and receive data from the Crazyflie 2.0. It has a specified range of over 1km within line of sight of the Crazyflie 2.0 and is just as the quadrotor, a fully open source project, allowing for firmware upgrades via USB and comes with an application programming interface (API) for the Python programming language.

2.1.2 Flow deck V2

The Crazyflie 2.0 offers the possibility to connect expansion boards to the headers, both on the top or on the bottom of the unit and are automatically detected and read by the power management MCU. The Flow deck V2 is an expansion board developed by Bitcraze AB that gives the Crazyflie the ability to discern when it is moving in any direction above the ground and acts as a complement to the IMU. It incorporates the VL53L1x time-of-flight sensor which measures the relative distance between the ground and the PMW3901 optical flow sensor that measures movement above the ground[4].

The Flow deck V2 is intended to be mounted to the bottom of the Crazyflie and allows the MAV to hold its position by itself without the need of an external positioning system. The way the optical flow works, is that it uses the visual data captured by the PMW3901 sensor and calculates a gradient from which it is possible to determine the distance and speed of which the craft is travelling. However, since optical flow depend on finding features on the plane of which the sensor is directed, it works best on matte surfaces. A surface that is very uniform or reflective will make it more difficult to match the features between each frame. This topic will be discussed in further detail in section 4.1.

2.1.3 Camera

In order to be able to localize the marker and determine the relative pose of the quadcopter, it is necessary to add an additional sensor. It is theoretically possible to detect and ascertain the position of a given target by using various approaches and thus, it may be of relevance to consider what features are important when choosing what type of sensor to use. To meet the conditions discussed in section 1.1.2 and 1.1.4, I opted for using a camera- and transmitter module from the company *Eachine*, sold as a spare part for their M80S RC Drone.

The module is very cheap, light weight and small, while still maintaining decent image resolution and build quality. However, because of the low price, it has some shortcomings and a professional grade component would probably be able to produce a much better end result.

Table 2.1: Camera- and transmitter module specifications

Total Weight	4 grams
Size (Width, Height, Depth)	(14.5, 12.0, 9.22) millimetres
Camera Resolution (Width, Height)	(640, 480) pixels
Camera Sensor	1/3 inch CMOS
Module Operating Voltage	2.9V - 5.5V
Video Output Format	PAL/NTSC
Transmitter Channels	48
Transmission Frequency	5.8GHz
Field of view (FOV)	140°

When mounting a camera that is to be used for visual odometry for a MAV, it is of great importance that it is attached in a rigid configuration in order to reduce vibrations and wobble that may introduce noise and/or other inaccuracies in the visual data. One should also consider that the field of view of the camera should not be occluded by the propellers or the craft itself. Also, the added mass may shift the center of mass of the MAV, which could alter the performance of the vehicle.

In order to keep the weight of the whole system to a minimum, as well as keeping the the whole construction as simple as possible, the camera and transmitter is powered directly from the Crazyflie 2.0. The camera- and transmitter module requires 2.9V-5.5V to operate, so it is possible to drive the module directly from the *VCOM(3.7V)* and *GND* pin-headers on the PCB of the quadcopter. Since the *VCOM* pin is only active when the Crazyflie 2.0 is turned on by the nRF51822 chip, it is also a very convenient solution because it eliminates the need of separate power management. However, this also lead to an adverse affect on the total amount of accessible energy for the Crazyflie 2.0, since the camera- and transmitter module draws about 1.3W during operation when powered with 3.7V. When adding any peripherals to the vehicle it is of interest to consider its power consumption both for operation and its contribution to the total amount of available energy by payload.

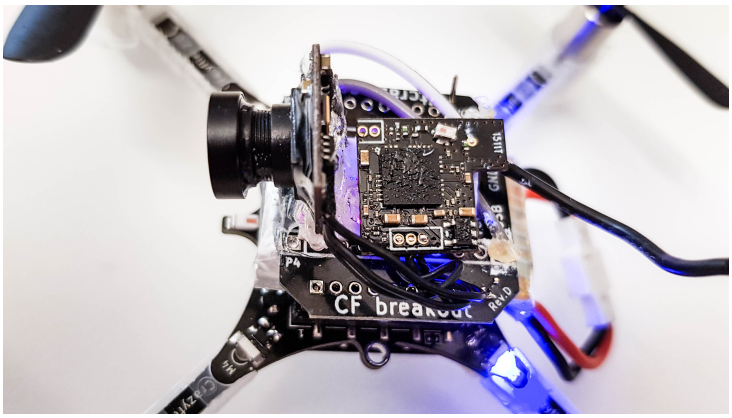


Figure 2.4: The camera and transmitter module was mounted on top of the Crazyflie 2.0 by soldering and attaching it onto a Breakout deck. More sophisticated solutions are available, but this is a quick and simple solution that offers great flexibility and sturdiness.

The camera- and transmitter module was soldered onto a *Breakout deck* for easy connection and disconnection. The Breakout deck is an expansion board provided by Bitcraze AB that allows for testing and development of hardware without the need of soldering or connecting it directly to the pin headers of the Crazyflie. This proved to be a suitable solution for the case of this module, as it also acts as a mounting platform, thus, we can dismiss the need of creating and designing a mounting platform for the module. The module was then fixated onto the Breakout deck simply by gluing it onto the surface of the deck with non-conductive glue.

This configuration will entail in the propellers occluding the view of the environment in the image plane when to rotors are stationary. However, this will not be a complication when the rotors are spinning at a high rate because of the aliasing effect that occurs. Because the frame rate of the camera is several times lower than the angular rate of the propellers, they will appear as invisible in the image frame during flight. The shift in the center of mass caused by the addition of the camera- and transmitter module is compensated for by the placement of the LiPo-battery.

Transmission and reception

Sources of radio transmission in the vicinity of the system may cause interference with the video transmission signal. If these signals occur on the same frequency band as the video feed, it could introduce noise which will reduce the signal-to-noise ratio (SNR), resulting in artefacts and noise in the video signal. This problem can be reduced by a proper channel selection on both the receiver and transmitter. The video transmitter and receiver used in this project follows an analogue convention, meaning that transmission can be made in real-time with very low latency and no image compression required. This is important because a latency in the range of only tens of milliseconds could impact the performance of the vision based control system drastically. However, because the received analogue signal must be converted into a digital format that we can process in the computer, there will always be a small delay between the time of transmission and when the broadcasted image reaches the framestore in the computer.

The broadcasted video signal is picked up by an *Eachine 5.8GHz OTG* USB video class (UVC) receiver. This particular receiver has 150 channels, so finding a relatively noise free channel for transmission and reception is an easy task. The receiver can be plugged directly into a free USB-port on the computer and requires no additional drivers to operate. The latency of the receiver is specified on the Eachine website to be at around 100ms and has a frequency range of 5.645GHz - 5.945GHz at a -90dBm sensitivity[8].

Chapter 3

Theory

This chapter aims to give the reader a basic understanding of the mechanics of a quadrotor MAV, as well as discuss the underlying theory behind transformation of points between coordinate reference frames, control theory and visual odometry. The contents of this chapter is considered a prerequisite for assimilating the context of the forthcoming chapters.

3.1 Basic mechanics of a quadrotor

Quadcopters have six degrees of freedom (6DOF), translational movement along the x, y and z -axes, as well as rotational movement along each of the three respective axes. The angular rotations along each axis are commonly referred to as: *Roll* (Φ), *Pitch* (Θ) and *Yaw* (Ψ) respectively. The quadcopter is lifted and propelled by four vertically oriented propellers, two of them rotating in a clockwise direction and the other set rotating in a counter-clockwise direction, as seen in figure 2.3. Each rotor has to support roughly one quarter of the total weight of the craft in equilibrium: $W_0 = \frac{1}{4}mg$, where m is the mass of the system and g is the gravitational constant. This allows us to calculate the operating speed for all four rotors, $\boldsymbol{\omega}_o$. However, this operating speed will produce a drag moment that each rotor has to overcome. Therefore, it is important that the motors are dimensioned such that they can produce enough torque to overcome this drag moment.

If we know the constant of proportionality, k_F between the force, $\overrightarrow{\boldsymbol{F}}_i$ generated by each rotor and its respective angular velocity $\boldsymbol{\omega}_i$, then the force generated by that motor can be expressed as:

$$\overrightarrow{\boldsymbol{F}}_i = k_F \boldsymbol{\omega}_i^2 \tag{3.1}$$

And if we know the constant of proportionality, k_M between the drag moment $\overrightarrow{\boldsymbol{M}}_i$ and the angular velocity, $\boldsymbol{\omega}_i$, then:

$$\overrightarrow{\boldsymbol{M}}_i = k_M \boldsymbol{\omega}_i^2 \tag{3.2}$$

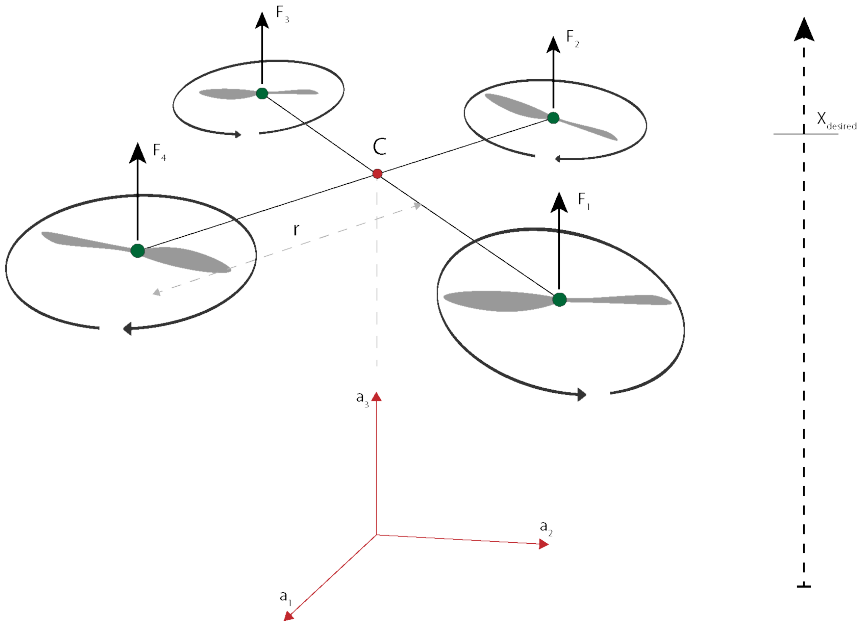


Figure 3.1: Basic model of a quadrotor. The force exerted by each motor is represented by $\boldsymbol{F}_i$ and r is the characteristic length of the system, i.e., the distance between the rotor and the center of mass. In an ideal one-dimensional model, the thrust output from the motors will determine the height, x of the quadrotor in the world frame.

This gives us the resultant force:

$$\overrightarrow{\boldsymbol{F}}_t = \sum_{i=1}^{4}(\overrightarrow{\boldsymbol{F}}_i) - mg\boldsymbol{a_3} \tag{3.3}$$

And if we know the distance to the center of mass between each rotor, we can calculate the moment around the center of mass such that:

$$\overrightarrow{\boldsymbol{M}}_t = \sum_{i=1}^{4}(r_i \times \overrightarrow{\boldsymbol{F}}_i + \overrightarrow{\boldsymbol{M}}_i) \tag{3.4}$$

, where r_i is the distance between an individual rotor, i and the center of mass, C. The first term inside the summation represent the moments due to the forces exerted by the rotors and the second term represent the reactions due to the angular direction of rotation of the rotors.

This means that for an ideal system in equilibrium, the thrust output from each motor is the same and they add up to support the total weight:

$$\sum_{i=1}^{4}(k_F\boldsymbol{\omega}_i^2) + mg = 0 \tag{3.5}$$

If we consider the mass, m and the gravitation, g as constant, any increase in angular velocity will yield an acceleration such that $\sum_{i=1}^{4}(k_F\boldsymbol{\omega}_i) + mg = m\boldsymbol{a}$, where $\boldsymbol{a} = \frac{d^2x}{dt^2} = \ddot{x}$ and x is the vertical position as seen in figure 3.1.

One of the reasons this is important is that when we design the control system for the quadcopter, we must take into consideration the limited capacity of the motors to generate thrust. If we let $u = \frac{1}{m}[\sum_{i=1}^{4}(k_F\boldsymbol{\omega}_i^2) + mg]$ be the control signal to the system, then:

$$u_{max} = \frac{1}{m}\left(\overrightarrow{\boldsymbol{F}}_{max} + mg\right) \tag{3.6}$$

, where $\overrightarrow{\boldsymbol{F}}_{max}$ is the maximum thrust that can be generated, determined by the peak motor torque.

3.2 Rigid transformations

Any position and orientation is associated with a reference frame and a rigid transformation is generally used to describe a geometric transformation between reference frames attached to different rigid bodies, whereas a displacement is essentially a transformation of points within a frame attached to a rigid body. This section will discuss how to transform points between reference frames in three-dimensional Euclidean space, $\mathbb{R}^3$.

A transformation of a point, $\mathbf{p}$, from a world frame coordinate system, $\mathcal{W}$, to a local coordinate system, $\mathcal{C}$, can be expressed as:

$$^{\mathcal{C}}\mathbf{p}_{\mathcal{W}} = {}^{\mathcal{C}}\mathbf{R}_{\mathcal{W}}\,{}^{\mathcal{W}}\mathbf{p} + {}^{\mathcal{C}}\mathbf{t}_{\mathcal{W}} \tag{3.7}$$

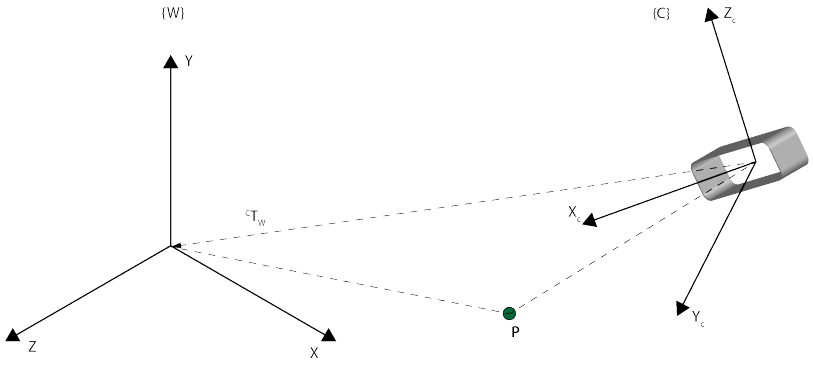


Figure 3.2: Transformation of a point, p, between coordinate frames

In figure 3.2 above, frame $\mathcal{W}$ indicate the world frame that is motionless and attached to the ground and frame $\mathcal{C}$ represent the body fixed frame that moves with the motion of the rigid body. Transformation of points between the two frames involve both *translation*, which can be described as the one-to-one corresponding translational displacement between two sets of points within a pair of coordinate systems, and *rotation*, which describes the angular displacement between the two frames.

3.2.1 Rotations

A rotation matrix, $\boldsymbol{R}$ must follow some requirements in order to be classified as a rotation matrix in the three-dimensional rotation group, often denoted as the special orthogonal group, **SO(3)**. By definition, the transpose of the rotation matrix must be equal to its inverse, i.e., $\boldsymbol{R}^T = \boldsymbol{R}^{-1}$. Also, since the column vectors of the rotation matrix are mutually orthogonal[34] the rotation matrix $\boldsymbol{R}$ is an orthogonal matrix, meaning: $\boldsymbol{R}^T\boldsymbol{R} = \boldsymbol{I}_{(3\times3)}$ Further more, the determinant of the rotation matrix must be equal to one: $det(\boldsymbol{R}) = 1$.

A rotation matrix in **SO(3)** has the general form:

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{r_1} & \boldsymbol{r_2} & \boldsymbol{r_3} \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{3.8}$$

, such that for a counter-clockwise (CCW) rotation about any linearly independent axis in three-dimensional space:

$$\boldsymbol{R_z}(\Psi) = \begin{bmatrix} \cos\Psi & -\sin\Psi & 0 \\ \sin\Psi & \cos\Psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.9}$$

$$\boldsymbol{R_y}(\Theta) = \begin{bmatrix} \cos\Theta & 0 & \sin\Theta \\ 0 & 1 & 0 \\ -\sin\Theta & 0 & \cos\Theta \end{bmatrix} \tag{3.10}$$

$$\boldsymbol{R_x}(\Phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\Phi & -\sin\Phi \\ 0 & \sin\Phi & \cos\Phi \end{bmatrix} \tag{3.11}$$

, where Φ, Θ, Ψ denote the angular rotation about the $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ axes respectively. These angles are considered *Euler Angles* as long as the condition of linearly independent axes is fulfilled and can be used to place a rigid body in $\mathbb{R}^3$. A single rotation matrix may also be formed by successive rotations about the linearly independent axes. Every rotation will then occur about the axes according to the sequence in which the rotation matrices are multiplied, e.g, following the sequence Z-Y-X will yield the following rotation matrix:

$$\begin{aligned} \boldsymbol{R}(\Psi,\Theta,\Phi) &= \boldsymbol{R_z}\boldsymbol{R_y}\boldsymbol{R_x} \\ &= \begin{bmatrix} \cos\Theta\cos\Psi & \sin\Phi\sin\Theta\cos\Psi - \cos\Phi\sin\Psi & \cos\Phi\sin\Theta\cos\Psi + \sin\Phi\sin\Psi \\ \cos\Theta\sin\Psi & \sin\Phi\sin\Theta\sin\Psi + \cos\Phi\cos\Psi & \cos\Phi\sin\Theta\sin\Psi - \sin\Phi\cos\Psi \\ -\sin\Theta & \sin\Phi\cos\Theta & \cos\Phi\cos\Theta \end{bmatrix} \end{aligned} \tag{3.12}$$

Rotations may also occur about an arbitrary axis, i.e., an axis other than the unit vectors representing the reference frame. In such cases, we need a way of determining the final orientation and the unique axis about which the point vector is rotated. This is often referred to as *Orientation Kinematics* [15]. A rotation of a generic vector $\boldsymbol{p}$ about an arbitrary axis $\boldsymbol{u}$ through an angle ϕ can be written as:

$$Rot(\boldsymbol{u},\phi) = \boldsymbol{I}\cos\phi + \boldsymbol{u}\boldsymbol{u}^T(1-\cos\phi) + \hat{\boldsymbol{u}}\sin\phi \tag{3.13}$$

, where $\hat{\boldsymbol{u}}$ is the skew-symmetric matrix representation of the vector $\boldsymbol{u}$ and is defined as:

$$\hat{\boldsymbol{u}} = \begin{bmatrix} 0 & -u_3 & u_2 \\ u_3 & 0 & -u_1 \\ -u_2 & u_1 & 0 \end{bmatrix} \tag{3.14}$$

This equation (3.13) is called *Rodrigues' rotation formula* and has proved to be especially useful during this project with the main reason being that by knowing the numerical values of the elements contained within the rotation matrix, it is possible to compute $\hat{\boldsymbol{u}}$ and ϕ by:

$$\hat{\boldsymbol{u}} = \frac{1}{2\sin\phi}(\boldsymbol{R} - \boldsymbol{R}^T) \tag{3.15}$$

and:

$$\cos\phi = \frac{\tau - 1}{2} \tag{3.16}$$

, where τ is the sum of the scalar values in the main diagonal of the rotation matrix, $\boldsymbol{R}$: $\tau = r_{11} + r_{22} + r_{33}$. However, we must consider that for $\tau = 3$ and $\tau = -1$, we will not be able to compute a unique axis of rotation because ϕ will either be equal to zero or π.

This same principle holds for when we want to calculate the Euler angles given a known rotation matrix. We will always have at least two solutions for every set of Euler angles for a given rotation matrix, and we may also have points of infinite solutions. To counter this issue, we need a second set of Euler angles to take care of the points of which we have infinite

solutions. Generally, for a rotation matrix, $\boldsymbol{R}_{zyx}$, its respective Euler angles, Ψ, Θ and Φ can be computed as:

$$\begin{aligned}
\Theta_1 &= -sin^{-1}(r_{31}) \\
\Theta_2 &= \pi - \Theta_1 \\
\Phi_1 &= atan2\left(\frac{r_{32}}{\cos\Theta_1}, \frac{r_{33}}{\cos\Theta_1}\right) \\
\Phi_2 &= atan2\left(\frac{r_{32}}{\cos\Theta_2}, \frac{r_{33}}{\cos\Theta_2}\right) \\
\Psi_1 &= atan2\left(\frac{r_{21}}{\cos\Theta_1}, \frac{r_{11}}{\cos\Theta_1}\right) \\
\Psi_2 &= atan2\left(\frac{r_{21}}{\cos\Theta_2}, \frac{r_{11}}{\cos\Theta_2}\right)
\end{aligned} \tag{3.17}$$

However, this can only be done if $r_{31} \neq \pm 1$ because it would mean that Ψ and Φ cannot be uniquely determined. In this case, we let $\Psi = 0$ and perform two calculations for two different cases; if $r_{31} = -1$:

$$\begin{aligned}
\Theta &= \frac{\pi}{2} \\
\Phi &= \Psi + atan2(r_{12}, r_{13})
\end{aligned} \tag{3.18}$$

, otherwise:

$$\begin{aligned}
\Theta &= -\frac{\pi}{2} \\
\Phi &= atan2(-r_{12}, -r_{13}) - \Psi
\end{aligned} \tag{3.19}$$

The above equations, (3.18 and 3.19), will give infinite solutions when $\Theta = \pm\frac{\pi}{2}$, hence, we let $\Psi = 0$ for the sake of convenience.

3.2.2 Translations

By neglecting the second term in the right hand side of equation 3.7, we assume that the two coordinate systems have a shared origin and that the transformation simply occurs without a translational difference between the two reference frames. Thus, in order to compensate for this difference, we must add the second term, denoted by ${}^{\mathcal{C}}\mathbf{t}_{\mathcal{W}}$ in the same equation. This is a vector containing information about the relative distance between the two frames, along each of the three respective axes, which can be interpreted as an addition of a constant to every point in the reference frame, or a shift of the origin between the two reference frames. The combination of rotation and translation as seen in equation 3.7 is called *rigid motion* or *rigid transformation*.

3.3 Visual Odometry

As we take an image of an object in the three-dimensional world around us, that object gets defined in a two-dimensional plane, meaning that we lose the third dimension in the process. Also, when we take a picture, it matters how the camera is oriented and positioned towards the world because various orientations and positions will yield different projections onto the image plane. *Visual odometry* is the process of resolving the pose of the camera, or in this case the MAV, in terms of position and attitude from data associated with images taken by the camera. This section will cover the theory behind this process and discuss the basic functionality and properties of the camera, as well as camera calibration and how the digital images are interpreted and processed by the computer.

3.3.1 Camera optics and image formation

A typical modern digital camera requires two essential elements, the imaging sensor and a lens. The imaging sensor is usually in the form of a *charge-coupled-device* (CCD) or a *Complementary Metal Oxide Semiconductor* (CMOS) -sensor, which are both light sensitive semiconductor devices with the primary function of detecting and transmitting information in order to form an image. The lens allows for the formation of an image onto the imaging sensor from the collection of rays of the entering light.

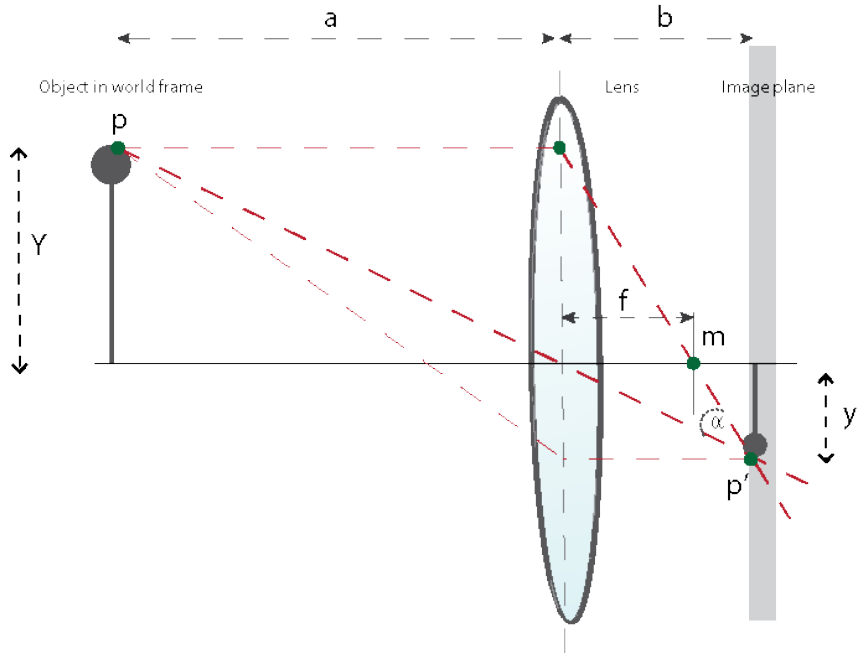


Figure 3.3: The thin lens model. Rays from a point, **p**, on an object converge on a point, **p'**, on the image plane.(a = distance between lens and object, b = distance between image plane and lens, **m** = focus point, f = focal length, Y = height of object in world frame, y = height of object in image plane)

The optical properties of the camera govern the relationship between the scene in the world frame and the projected image on the image plane [37]. The *thin lens model* shown in figure 3.3 above is an approximative mathematical model where rays of light emitted from a point of an object travel along a path through the lens, converging onto another point on the image plane. Through this model, the projection on the image plane is a function that depend on two factors; the distance between the image plane and the lens, denoted as b, and the distance between the lens and the object in the world frame, denoted as a. The axis going through the center of the lens is called the *optical axis*, and the plane that lies perpendicular to that axis is called the *focal plane*, which is centered at the *optical center*. All the rays that are going parallel to the optical axis of the lens go through a point, **m**, defining the focus of the lens, and thereafter hits the image plane where the image is projected.

When the two rays emitted from the point, **p**, intersect at the image plane on point **p'**, it can be proved that:

$$\frac{1}{f} = \frac{1}{a} + \frac{1}{b} \tag{3.20}$$

, where f defines the focal length of the lens. At the conditions where this equation (3.20) is true, that is where the projected image is at its theoretically sharpest representation of the object in the world frame. This relationship expressed in equation 3.20 is called the *fundamental equation of the thin lens*.

So by controlling the parameter b, we can control the focus by changing the distance between the lens and the image plane. Another way to focus the image would be to move the lens and the image plane closer or further away from the object in the world frame, thus, controlling the parameter a. In modern digital cameras, this process is done automatically (by adjusting b), through a procedure called *autofocusing*. If we look at the size of the object in the world frame in relation to the projection of that same object in the image plane, we can directly relate them through similarity of triangles, such that:

$$\frac{Y}{a} = \frac{y}{b} \tag{3.21}$$

, where Y is the size of the object in the world frame and y is the size of the object in the image plane. This tells us that the size of the object in the image plane will change in relation to the relative distance between the two. However, if the point $\mathbf{p}$ moves along a ray, keeping the angle α as seen in figure 3.3 constant, the perspective projection will also remain constant and we can therefore not without ambiguity determine where an object is located in the world frame from a single point in the image plane, but we require *at least* two points in order to be able to reason about the position of an object in the world frame, as we will discuss in further detail in the following sections.

3.3.2 Homogeneous coordinates and the projective plane

As we perceive the world from a first person view, we are at the origin of three-dimensional space and everything around us is measured relative to that origin. This means that any parallel horizontal lines in physical space will be observed as converging to a single point. This concept holds true especially in the image plane. These points are referred to as *vanishing points*, indicated by $\mathbf{V}_1$ and $\mathbf{V}_2$ in figure 3.4 below. Between these vanishing points lies the *vanishing line*, also known as the *horizon*. The concept of vanishing points and vanishing lines allows us to reason about the relative pose of the observer of the scene, in this case, the camera. If we know the actual height of an object in physical space, we could also use this concept to reason about its movement in the image plane.

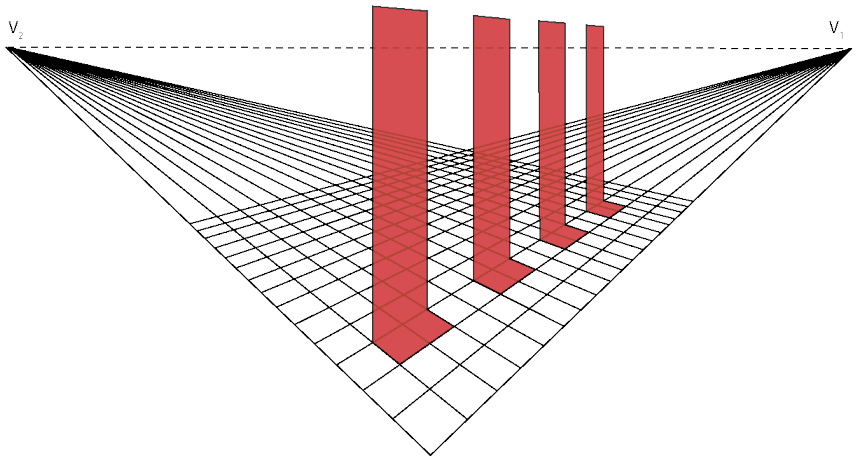


Figure 3.4: Illustration of the concept of vanishing points

Homogeneous coordinates can be especially useful when representing points at infinity, such as vanishing points from the three-dimensional world projected onto the two dimensional image frame. If we were to imagine that each point on the image plane can be seen as a ray from the observer origin, then all points on that ray can through geometric intuition be seen as essentially the same, such that:

$$(x, y, 1) \cong (wx, wy, w) \tag{3.22}$$

, so any point in $\mathbb{R}^3$ along that ray will project to the same point on the image plane, meaning that the relevance lies not in the distance of the point along it, but in the direction of the ray.

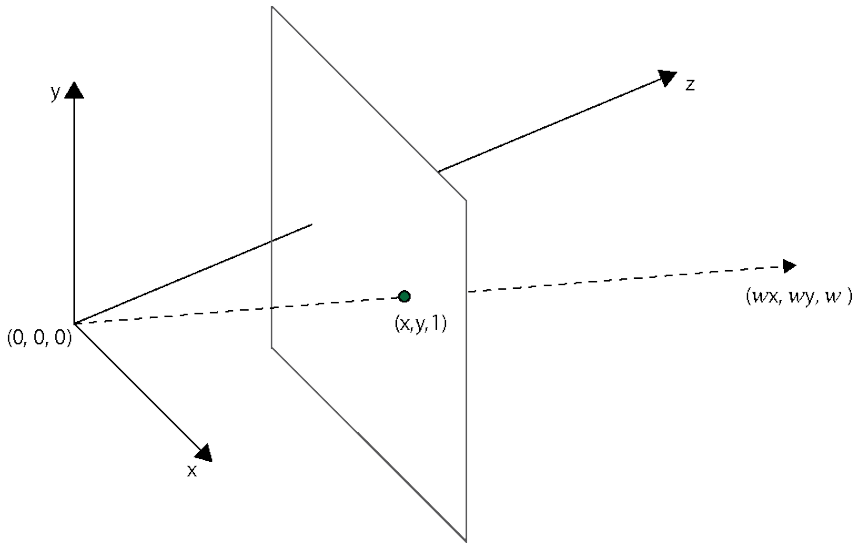


Figure 3.5: Geometric intuition: A point on the image plane is a ray in projective space. Each point, *(x, y)* on the image plane is represented by a ray, *(wx, wy, w)*.

So when we want to represent a two-dimensional vector in the image plane by a three-dimensional vector, we append an additional element to every 2D-point, such that:

$$\begin{bmatrix} x \\ y \end{bmatrix} \Rightarrow \begin{bmatrix} x \\ y \\ w \end{bmatrix} \tag{3.23}$$

, where w is a scaling factor. In fact, every representation of an n-component vector is through the *homogeneous coordinate representation*, represented as an $(n+1)$-component vector [15]. If we suppose that the image plane of the camera lies at $w = 1$, then the ray through the pixel (x, y) can be represented in homogeneous coordinates by: $(x, y, 1) \sim (wx, wy, w)$. This can be done for any value of the scaling factor, w, provided $w \neq 0$ and $w \neq \infty$. However, if $w = 0$, then we would essentially describe a point that lies at infinity and does not correspond to any finite pixel in the image plane. By adding such points in a two-dimensional image, we form the vanishing line in the image.

3.3.3 Camera projection

Camera calibration is the process of estimating the cameras internal characteristics and the camera position and orientation in the world, commonly known as the *intrinsic* and *extrinsic* parameters respectively. The intrinsic parameters of the camera include parameters such as the focal length of the lens, the optical center, also known as the *principal point*, the pixel skew and the optical distortion in the image. Knowing the intrinsic parameters of the camera is essential in order to with good accuracy, be able to estimate the extrinsic parameters, i.e., the relative camera position and orientation. As discussed in section 3.3.2, camera projection involve mapping points in the physical three-dimensional space onto the two-dimensional imaging plane. Through similarity of triangles, it can be proved that for a given point $\boldsymbol{x} = (u_x, u_y)$ projected on the image plane and the corresponding point in three-dimensional space, $\boldsymbol{P}(X, Y, Z)$, that:

$$\frac{f}{Z} = \frac{u_x}{X} = \frac{u_y}{Y} \tag{3.24}$$

which gives us:

$$\begin{aligned} u_x &= f\frac{X}{Z} \\ u_y &= f\frac{Y}{Z} \end{aligned} \tag{3.25}$$

, where f is the focal length. From these equations (3.25), we can express the point $\boldsymbol{x}$ using homogeneous coordinates when magnified by a factor λ, i.e., the distance to the camera, by:

$$\lambda \begin{bmatrix} u_x \\ u_y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{3.26}$$

Also, if the origin of the two-dimensional image plane does not coincide with where the principal axis lies, i.e., where the optical axis intersects the image plane, it is necessary to translate the point $\boldsymbol{x}$ to the desired origin. An image in a computer is usually defined as a $(m \times n)$-matrix, hence, the origin of the image is defined at the top left corner, whereas in the optical image, the origin is defined at the center of the image. Let this translation determined by the principal point be defined by (c_x, c_y). This gives us:

$$\begin{aligned} u_x &= f\frac{X}{Z} + c_x \\ u_y &= f\frac{Y}{Z} + c_y \end{aligned} \tag{3.27}$$

, which can be expressed in matrix form as:

$$\lambda \begin{bmatrix} u_x \\ u_y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{3.28}$$

In order to convert the point $\boldsymbol{x}$ from a measure in pixels in two-dimensional space, to a measure in centimetres in two-dimensional optical space, we need to know the resolution of the camera in pixels/centimetre, as well as the magnification caused by the focal length f. If we let α_x and α_y define the pixel scaling factor in $x-$and y-direction, then to measure the point $\boldsymbol{x}$, its $x-$and y-coordinates should be multiplied with α_x and α_y respectively. This gives us:

$$\lambda \begin{bmatrix} u_x \\ u_y \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha_x f & sf & c_x \\ 0 & \alpha_y f & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{3.29}$$

, where s represent the *slant factor*, used when the image is not normal to the optical axis.

Putting it all together, the camera projection matrix for a first person view camera configuration can be expressed as:

$$\lambda \begin{bmatrix} u_x \\ u_y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{I_{3\times 3}} & 0 \end{bmatrix} \begin{bmatrix} X_{camera} \\ Y_{camera} \\ X_{camera} \\ 1 \end{bmatrix} \tag{3.30}$$

, where the first term on the left hand side of the equals-sign represent the 2D-pixel values in pixel-domain and the last factor on the right hand side of the equation, $[X_{camera}, Y_{camera}, Z_{camera}, 1]^T$ represent the first person view in three-dimensional physical space.

Now, in the case of this project, we want to be able to recover both the camera orientation relative to the world, as well as the translational offset by looking only at the image itself, so

if we want to map this image between different perspectives, we utilize the previously discussed transformation of points between coordinate systems covered in section 3.2. Using a rotation matrix and a translation vector, we can through a *homogeneous transformation* describe the point $\boldsymbol{x}$ from one frame to another:

$$\boldsymbol{x} = \boldsymbol{K} \begin{bmatrix} \boldsymbol{R_{3\times 3}} & \boldsymbol{t_{3\times 1}} \\ 0 & 1 \end{bmatrix} \boldsymbol{X_{camera}} \tag{3.31}$$

, where $\boldsymbol{K}$ is the camera matrix containing the intrinsic parameters. Through equation 3.31, we can recover the rotation matrix and the translation vector: $\boldsymbol{R} \in SO(3), \boldsymbol{t} \in \mathbb{R}^3$ if the $\boldsymbol{K}$-matrix is known through calibration.

3.3.4 Camera calibration

Before the use of computers, if the focal length of the lens (in millimetres) and the dimension of the camera film was known, the $\boldsymbol{K}$-matrix (as shown in equation 3.31) was computed manually. Today we can quite easily compute the intrinsic parameters of the camera with the use of computers. The process usually involves a provision of a set of images containing a calibration pattern used as a point of reference or measure for the computer. The properties of the pattern in physical space must be known. The intrinsic and extrinsic parameters can then be computed by a *least-square* minimization method[33].

Cheap cameras with a large field of fiew (FOV) such as the one employed in this project (see section 2.1.3 and 5.4) often suffer from distortion in the image. This distortion come in the form of *radial distortion* and *tangential distortion*. Radial distortion appears when the pixel point is distorted proportionally to the radius from the center of the images. This can be seen as straight lines in physical space appearing as curved in the image frame. Radial distortion is typically modelled as:

$$\begin{aligned} x_{dist} &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6 + ...) \\ y_{dist} &= y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6 + ...) \end{aligned} \tag{3.32}$$

, where k is the unknown parameter that we want to find through calibration and r is the radius from the center of the image, i.e., $r^2 = u_x^2 + u_y^2$. Typically, using only two coefficients are sufficient for most calibration purposes, but for severe distortion, additional coefficients may be used (in our case, we compute three coefficients for the radial distortion). So for each pixel with coordinates (x, y), the position is corrected by (x_{dist}, y_{dist}) [24].

The tangential distortion occurs when the elements of lens does not perfectly align with the imaging plane and can be corrected by:

$$\begin{aligned} x_{dist} &= x + [2p_1 xy + p_2(r^2 + 2x^2)] \\ y_{dist} &= y + [p_1(r^2 + 2y^2) + 2p_2 xy] \end{aligned} \tag{3.33}$$

So through equation 3.32 and 3.33, we have a set of distortion parameters which can be computed by following the procedure covered and documented in [35] and [39]. The results of calibrating the camera used in this project can be found in section 5.1.

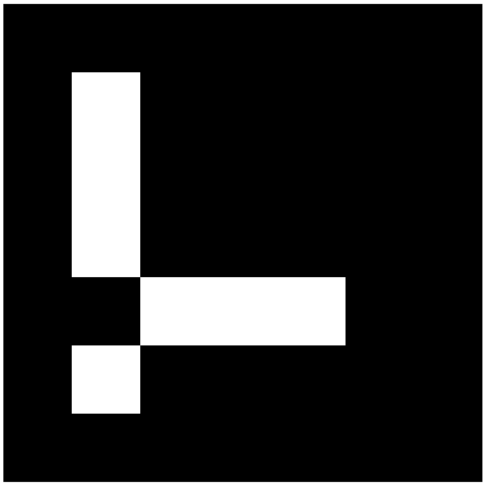
(a) Original image, unrectified (b) Rectified image

Figure 3.6: Comparison of an unrectified versus rectified image from calibration

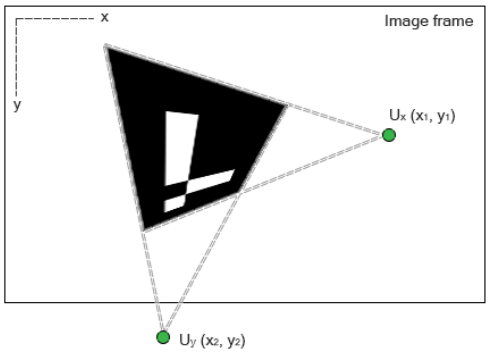
3.4 ArUco library and relative pose estimation

In this thesis, the *ArUco* module developed by Rafael Muñoz and Sergio Garrido[31][13] was employed as the library for target detection and pose estimation. The library is composed of a set of dictionaries of different markers, each with its own unique identifier(ID) but with the same characteristic design. An ArUco marker is a synthetic square planar marker with an inner binary identifier matrix contained within a thick black border. Similar to a barcode or a QR-code, each marker has its own unique identification which is determined by an inner binary matrix.

The AruCo library is composed of several predefined dictionaries with predefined sizes, meaning that the internal binary matrix may be of size e.g., $(4 \times 4), (5 \times 5)$ or (6×6). The original ArUco dictionary utilize a (5×5) internal binary matrix where the 1st, 3rd and 5th column represent the parity bits of the marker. The parity bits are composed in *Hamming code* such that it can detect up to two-bits error or correct a one-bit error by checking for odd- or even parity. If the parity does not conform to the defined parity, the transmission is treated as erroneous. The rest of the columns represent the data bits where a white part represent a logical one and a black section in the inner matrix represent a logical zero. In the original ArUco dictionary, since there are two columns with a total of five elements, a total of ten bits are allocated for data. Thus, the maximum numbers of markers that can be encoded in the dictionary is $2^{10} = 1024$ different markers. The identification of the marker is decoded by reading the 2nd and 4th column from top to bottom, left to right, such that for the marker in figure 3.7 below, the $ID = 12$.



(a) ArUco marker from the Original dictionary (5× 5). The identification of this marker is read as: 0000001100 in binary, which yields 4 + 8 = 12 in decimal, hence, the ID of this marker is 12.



(b) We must have at least two perpendicular vanishing points to be able to compute the relative rotation

Figure 3.7: ArUco marker from Original dictionary.

Now, in order to perform a pose estimation for the origin of the reference system of the camera relative to the reference system of the marker, we want to know all the relative angles in $\mathbb{R}^3$, as well as the translation between the two reference systems. One possible way to determine the relative angles is through the concept of vanishing points. Through finding at least two perpendicular vanishing points of the marker, its relative rotation can be computed if the intrinsic parameters of the camera is known by multiplying the inverse of the camera matrix, $\boldsymbol{K}$, with the respective coordinates of the vanishing points in the image plane:

$$\boldsymbol{R} = [\boldsymbol{r_1}, \boldsymbol{r_2}, \boldsymbol{r_3}] \in SO(3) \tag{3.34}$$

$$\begin{aligned} \boldsymbol{r_1} &= \frac{\boldsymbol{K}^{-1} u_x}{||\boldsymbol{K}^{-1} u_x||} \\ \boldsymbol{r_2} &= \frac{\boldsymbol{K}^{-1} u_y}{||\boldsymbol{K}^{-1} u_y||} \\ \boldsymbol{r_3} &= \boldsymbol{r_1} \times \boldsymbol{r_2} \end{aligned} \tag{3.35}$$

, where u_x and u_y represent the respective coordinates of the vanishing points in the image plane as seen in figure 3.7b. The pan- and tilt angles of the cameras can then be computed as: $\alpha = tan^{-1}(\frac{r_3(1)}{r_3(3)})$, $\beta = cos^{-1}(r_3(3))$. However, since we must be able to recover both the camera orientation relative to the marker, as well as the translation by looking only at the image itself, we utilize a *homography transformation* from a planar object in the physical three-dimensional space to a planar surface in two dimensions.

As such, if we define a point in planar two-dimensional space as: $\boldsymbol{x} = [x, y, 0, 1]$, which is projected onto the image frame to a point: $\boldsymbol{m} = [u_x, u_y, 1]^T$ in homogeneous coordinates, we can see that we have eliminated the third column, $\boldsymbol{r_3}$ in the rotation matrix and thereby obtained a two-dimensional homography transform where we have:

$$\lambda \boldsymbol{m} = \boldsymbol{K}[\boldsymbol{R}, \boldsymbol{t}] \begin{bmatrix} u_x \\ u_y \\ 1 \end{bmatrix} \tag{3.36}$$

In this way, we transform the x,y-coordinates in the planar world to a set of corresponding x,y-coordinates in the image plane:

$$\lambda \boldsymbol{m} = \tilde{\boldsymbol{H}} \boldsymbol{x} \tag{3.37}$$

, where $\tilde{\boldsymbol{H}} = \boldsymbol{K}[\boldsymbol{r_1}, \boldsymbol{r_2}, \boldsymbol{t}]$. The homography matrix, $\tilde{\boldsymbol{H}}$, can be estimated through the *ArUco* library with the help of projective transformation by taking four points in the marker frame and compute a correspondence to them in the image plane. These points represent the four corners of the given ArUco marker and through this four point correspondence, the rotation matrix, $\boldsymbol{R}$ and the translation vector, $\boldsymbol{t}$ can be computed numerically. This is done by taking the matrix $\tilde{\boldsymbol{H}}$ estimated from the four points in physical space to the image plane such that:

$$\boldsymbol{H} = \boldsymbol{K}^{-1} \tilde{\boldsymbol{H}} = [\boldsymbol{r_1}, \boldsymbol{r_2}, \boldsymbol{t}] \tag{3.38}$$

and

$$\boldsymbol{r_3} = \boldsymbol{r_1} \times \boldsymbol{r_2} \tag{3.39}$$

, where $\boldsymbol{H}$ is the transformation from $\mathbb{P}^2$ to $\mathbb{P}^2$

So if we are able to detect the four corners of the marker in an image taken by the camera, we can estimate the pose of the camera relative to that marker. In order to properly detect the marker, it should have some white space around the black border encompassing the inner matrix. The reason behind this is that the detection algorithm that the ArUco library utilize cannot disambiguate the marker from its surroundings if the black border is not clearly contrasted from its environment. The pose estimation algorithm in the ArUco library outputs two vectors once it has detected the four corners of the marker in the image through a thresholding process. These two vectors are the rotation vector and the translation vector, each containing three parameters; the relative rotation between the two reference frames and the translation between them. These six parameters are transformed though *Rodrigues formula* as seen in equation 3.13 and a homogeneous transformation such that we get:

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_m \\ y_m \\ z_m \\ 1 \end{bmatrix} \tag{3.40}$$

, where $[x_c, y_c, z_c, 1]^T$ is the homogeneous representation of the camera frame reference system, the (4×4)-matrix is the transformation matrix and $[x_m, y_m, z_m, 1]^T$ is the representation of the marker reference frame. So once we know the transformation matrix that relates these two systems, we can use these parameters as input to the control system for controlling the position and orientation of the quadcopter.

3.4.1 Feature based detection and the Direct Method

The ArUco library uses a filter based method for detecting the marker in the scene, but one should also acknowledge that other methods exist. Key-frame based detection methods, such as ORB, SIFT and SURF can prove to be reliable methods for detecting objects in the image frame and they build upon matching features between different image frames through 2D to 2D correspondences. From this correspondence it is possible to extract information about the 3D-representation of the camera relative to the world and obtain a pose estimation of the camera. For example, [6] presents a monocular SLAM-based visual odometry method for stabilizing the Crazyflie 2.0 and provide autonomous flight by utilizing a key-frame based approach.

Feature-based detection relies on finding *features* in images, i.e., finding regions in the image which exhibit maximum variation when shifted or rotated in any direction by a small amount. Once the features in the image have been detected, each feature is given a *description* in order to be able to map the features from one image to another. The description basically describes the region around each feature, making it possible to find the same feature in two images and align them. The feature-based detection algorithm evaluated in this thesis is *ORB* (Oriented FAST and Rotated BRIEF), which builds upon the FAST keypoint detector and the BRIEF descriptor[32] and comes available in the OpenCV library. The ORB keypoint detector and descriptor is marketed as an efficient free alternative and improvement over the patented algorithms SIFT and SURF which either impose a high computational cost or yield poor approximations[32]. However, after evaluation and comparison of ORB and the ArUco library (see section 5.4.1), I decided on using the ArUco library which comes with an available detection and pose estimation function readily available, hence, the rest of this thesis will primarily focus on detection using the direct method.

Detection of ArUco markers

The process of detecting the markers in the images taken by the camera is comprised of two main steps. The first step involves image segmentation through thresholding and extraction of contours so that we can find square shaped candidates of markers in the image. Those rectangular shapes found in the thresholded image that do not approximate to a square gets disregarded and we proceed to the second step. Once we have found a set of candidates for markers in the image, we analyse the inner binary matrix of each candidate by extracting the white and black bits to determine which of them are valid markers. This is done through computing the homography matrix and thresholding using the *Otsu's method* [31] so that perspective projection is removed and we get a binarized image containing a grid of binary values. The bits are then analysed to determine which of the specific dictionaries the marker belong to. Each candidate holds four possible identifiers which correspond to four possible rotations.

3.5 Control

This section will discuss some basic principles of control theory, which is the study of a dynamical systems reaction to a given input over time. The controller of an MAV is an essential component because the vehicle must be able to compute its control demands to each rotor based on its current position and orientation, as well as the target position and orientation in real-time. The general idea when designing a control system is that the controller should be able to compute an *error signal* from a measured system state value and a desired target state. In a closed-loop system, the difference between them is then applied as a feedback to the input of the controller such that the error signal will eventually converge exponentially to zero. One should also consider that a well designed controller should be able to perform this process at a certain rate whilst sustaining stability.

When evaluating the response of the system, one could look at a numerous amount of parameters related to its behaviour in both time domain and frequency domain. However, in order to determine all the physical parameters of the system, it is often necessary to create a mathematical model of the system, describing all its inertial properties, drag coefficients and thrust maps, etc., from which we can perform a number of simulations. This can prove to be a very time consuming process, especially when dealing with complex dynamical systems. An extensive documentation of the system parameters of the Crazyflie 2.0 has been made previously by *Julian Förster* through measurements, calculations and experiments [12], Hence, for the rest of this thesis, we will concentrate on a non-model based approach and analysis in time domain.

3.5.1 PID control

In this project, a set of closed-loop controllers are used to continuously compare the measured output state of the system with the desired state, producing an error signal, $e(t) = x_{desired}(t) - x_{actual}(t)$, and convert it into a control action to reduce the error. The error signal $e(t)$ may arise as a result of change in the process being controlled or because the desired set point (SP) has changed and will always exponentially converge to zero if there exists constants, α, β and time t_0 such that for all instants of time $t \geq t_0$:

$$||e(t)|| \leq \alpha e^{-\beta t} \tag{3.41}$$

A closed-loop controller differ from an open-loop controller in that a feedback from the output of the process being controlled is used to modify the input signal such that the system maintains the desired state, whereas in an open-loop controller, the input to the controller is independent of the system output.

The proportional-integral-derivative controller (PID controller) is a type of closed-loop controller and is one of the most frequently used controllers in the industry. As its name suggest, the PID controller consists of three correcting terms; a *proportional* part, a *derivative part* and an *integrating* part. These terms are summed to calculate the output of the PID controller, $u(t)$, which is sent as input to the system. The system that we wish to control is commonly referred to as a *plant* or *process*. A schematic representation of a PID controller with a feedback loop can be seen in figure 3.8 below.

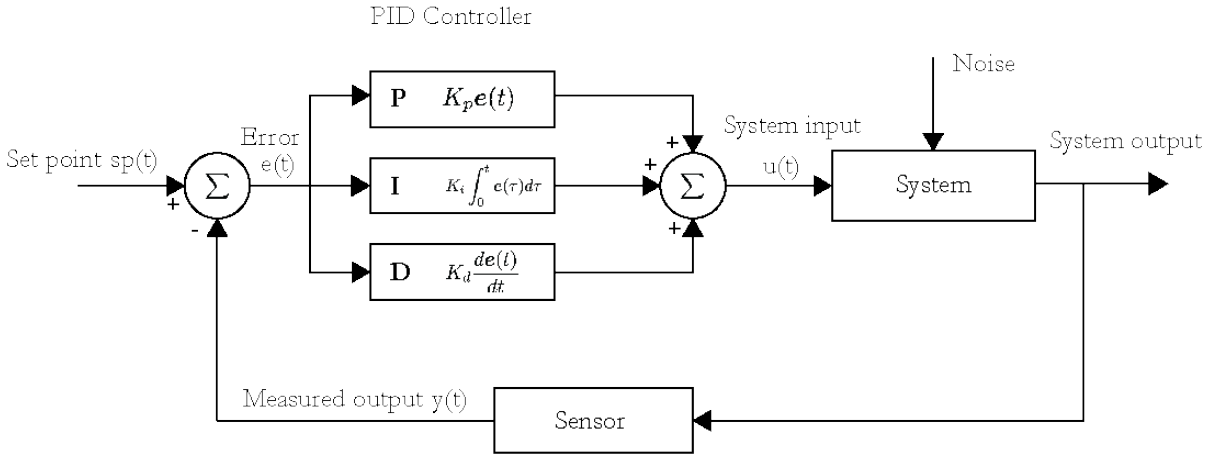


Figure 3.8: Block diagram of a PID controller in a feedback loop. $sp(\mathrm{t})$ is the desired set point, $e(\mathrm{t})$ is the error signal, $u(\mathrm{t})$ is the output from the PID controller and $y(\mathrm{t})$ is the measured value of the process being controlled.

When combining all three terms, the equation describing the action of the controller can be written as:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t) \tag{3.42}$$

, where K_p is the proportionality constant, K_i is the integral constant, K_d is the derivative constant and τ is the variable of integration. Through taking the *Laplace transform* of equation 3.42, we get:

$$G_{PID}(s) = K(1 + \frac{1}{T_i s} + T_d s) \tag{3.43}$$

, where K is the gain constant, T_i is the integral time and T_d denote the derivative time.

The **proportional** part of the control signal is directly proportional to the size of the error, meaning that the gain K_p governs the reaction to the current error. Generally, an increase of the proportional gain will allow for a faster control system. However, if it is set too high, the process will exhibit oscillations and may eventually become unstable.

The **integral** component of the PID controller reacts to the accumulation of recent errors over time. This means that the integral response will increase over time unless the error remains at zero, thus, driving remaining difference between the set point and the state of the process to zero. The remaining difference between the set point and the state of the process is referred to as the *steady-state error*. For a process that is in it self non-integrating, meaning that it accumulates its input up until it has reached a point of equilibrium, a controller with an integral part is required in order to fully drive the steady-state error to zero in the presence of noise[36].

The **derivative** part of the controller output is proportional to the rate of change with time of the error signal and can be thought of as the damper of the control signal, with the purpose of regulating the speed of the system and increase its stability. However, because the derivative part looks at the slope of the error signal over time, if its respective constant of proportionality, K_d, is set to high, the system may become very sensitive to high frequency noise and may cause the system to become unstable. As such, careful tuning of all three parameters, K_p, K_i and K_d must be performed in order to have the system behave in a favourable way.

PID Gain Tuning

Finding optimal values of the proportional constants for a system that we have no direct knowledge of is not a trivial task. If we know the transfer function of the system that we want to control, we could simulate the system with various gain values and evaluate the response from the given set of parameters. However, this requires a complete knowledge of the real system and the environment in which it operates in because the response of any physical system will in some degree deviate from the ideal system simulation. Since we are adopting a non-model based approach, we will resort to finding the gain values from analysing the system output as a function of time from experiments.

Various tuning methods have been proposed since the formulation of the PID controller was presented by *Nicholas Minorsky* in 1922[5] and an example of two commonly known heuristic tuning methods are the *Ziegler-Nichols method* and the *AMIGO*-method. Both of these methods can be implemented in two different ways, either through gradually increasing the proportional gain until the system begins to oscillate and then analyse the system output, or by applying a unit step to the system and evaluate the system step response. Through the step response, we can get information about the stability of the dynamical system as well as its ability to reach a stationary state when a given input signal is applied. Note that directly applying these methods may not yield parameter values that give us perfect regulation, but some manual tuning is often required in order to have the system behave in a desirable way.

In order to dimension the gain parameters, we need to define some metrics that can be used to characterize a well-regulated system. Firstly, the most basic requirement of the controller is *Stability*, which means that the error signal, $e(t)$, must at some point converge to zero (as described in equation 3.41) and exhibit minimal oscillations. Secondly, we want the system to behave in a desirable way with regard to external disturbances and reach the commanded set-point within the required time and minimize the steady state error. We can model the response of the controller through the following metrics when the system is subject to a step reference change:

- L: *Dead Time*, i.e., the time delay from when the controller output is issued to when the process begins its initial response.
- $T_{63\%}$: Time constant defined at the elapsed time from the beginning of the step, to 63% of the change in reference, after the dead time, L has elapsed.
- $K = \frac{\Delta y}{\Delta u}$: *Static gain*, i.e., the ratio between the system output, y and the constant input, u at steady state.

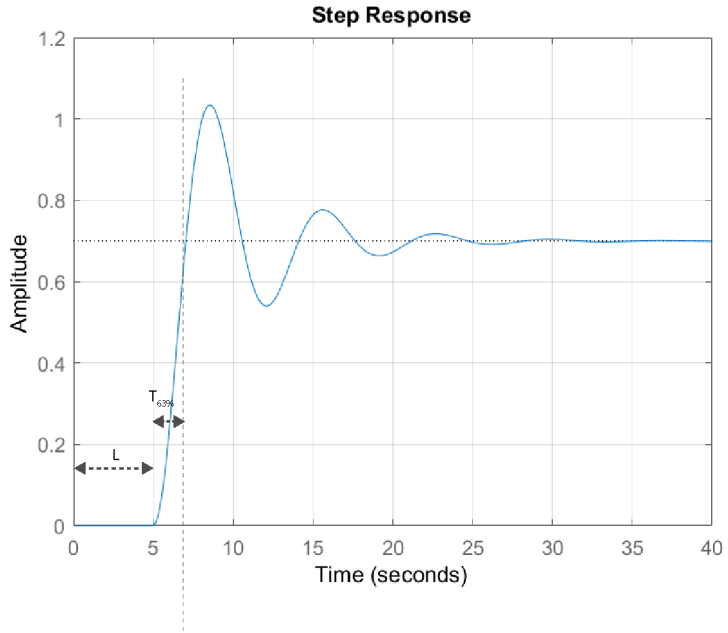


Figure 3.9: Example step response of a system with transfer function: $H(s) = e^{-5s}\frac{1.4}{2.4s^2+s+2}$, indicating the time constant, $T_{63\%}$ and dead time, L.

Before we start tuning the PID controller, we should also define the intentions of the controller in terms of how we want the system to behave by:

- **Rise Time**: The time required for the error signal, $e(t)$, to reach between 10% and 90% from the initial step to a steady state.
- **Overshoot**: The peak magnitude of the error signal, $e(t)$, after the initial breach of the reference step. Commonly denoted in terms of percentage.
- **Steady State Error**: The magnitude of the error signal, $e(t)$ after the system has converged to its steady state.
- **Settling time**: The elapsed time between the initial system response and the steady state response which remain within $\pm 5\%$ in terms of magnitude of the final system output.

Ziegler-Nichols Method

In the 1940s, *John G. Ziegler* and *Nathaniel B. Nichols* proposed a set of rules for determining the values of the proportional constants, K_p, K_i and K_d through analysis of the transient step response of a plant [5]. Through knowledge of the parameters, K, L and $T_{100\%}$ as described in section 3.5.1, the respective gains can through the *Zieger-Nichols method* be calculated as:

Table 3.1: Calculations of parameters for a PID controller from *Ziegler-Nichols* step response method:

K	T_i	T_d
$\frac{1.2T_{100\%}}{KL}$	$2L$	$\frac{L}{2}$

, where T_i is the integral time and T_d is the derivative time (as in equation 3.43), such that:

$$\begin{aligned} K_p &= K \\ K_i &= \frac{K}{T_i} \\ K_d &= K \cdot T_d \end{aligned} \tag{3.44}$$

Note that *Ziegler-Nichols* method use the time parameter $T_{100\%}$, which denote the time of which it takes the step response to reach 100% of the reference change, rather than 63%.

The AMIGO Method

AMIGO is an abbreviation for *Approximate M-constrained Integral Gain Optimisation* and was presented by *T. Hägglund* and *K. J. Åström* in 2002 as an improvement to the Ziegler-Nichols method with significantly better performance. It is used to calculate the proportional parameters in a similar fashion to the procedure used in the Zieger-Nichols method and is applicable for any system that can be modelled as a first order system with dead time (FOPTD) or integrating with dead time [30][14]. Through measuring the parameters, K, L and $T_{63\%}$ from the step response of the system, the parameters for the process can be calculated as:

Table 3.2: Calculations of parameters for a PID controller from the AMIGO step response method:

k	T_i	T_d
$\frac{1}{K}(0.2 + 0.45\frac{T_{63\%}}{L})$	$\frac{0.4L+0.8T_{63\%}}{L+0.1T_{63\%}}L$	$\frac{0.5LT_{63\%}}{0.3L+T_{63\%}}$

, and the respective gains, K_p, K_i and K_d can subsequently be calculated following equation 3.44 by letting $k = K$.

3.5.2 Kalman filters

Through the ArUco library (discussed in section 3.4), the target marker can be detected and validated and the pose of the camera relative to that marker can be computed. The pose of the camera relative to the detected marker is then given as input to the control system so that the MAV may align itself such that the y-axis of the body fixed frame of the quadrotor lies collinear with the z-axis of the reference frame of the detected marker. However, because of the noise introduced by e.g., the limited capabilities of the camera, transmission losses, vibrations in the reference frame of the quadrotor or by loss of detection, we require a method for estimating the relative pose of the MAV given the incomplete or noisy data from the images taken by the camera.

The *Kalman filter*, named after its author, *Rudolf E. Kálmán* (1960)[38], is an effective recursive filter or algorithm that provides an optimal estimation of unknown variables from noisy or inaccurate measurement data, hence, the word *filter* amounts to not only cleaning up the measured data, but also projecting the measurement onto a state estimate. The Kalman filter addresses the problem of predicting the state of a dynamical system at discrete time, k, given measurements from the current state at time $k-1$ and its uncertainty matrix. This section will cover the basics of the *Linear Kalman Filter* as it is the one implemented in this project (see section 4.7.2), but note also that extensions and improvements to this method have been developed, such as the *Extended Kalman filter*(EKF) and the *Unscented Kalman filter*(UKF) which are used for non-linear systems where the standard Kalman filter is not sufficient.

The linear Kalman filter

The linear Kalman filter is a discrete time algorithm consisting of a prediction and a measurement update step which can be run independently from one another. If we assume a current state of our system, $\boldsymbol{x}_k$, containing elements describing its position, $\boldsymbol{p}$ and velocity, $\boldsymbol{v}$, as:

$$\boldsymbol{x}_k = (\boldsymbol{p}, \boldsymbol{v}) \tag{3.45}$$

,where subscript k indicate the current discrete time step, then the Kalman filter will assume a correlation between the elements contained in $\boldsymbol{x}_k$, captured by a covariance matrix, $\boldsymbol{P}_k$. It will then produce an estimate, $\hat{\boldsymbol{x}}_k$ from the previously estimated state, $k-1$, the covariance matrix and the current measurement from time step k for the conditional probability that the error between the measured and estimated state are Gaussian distributed with a mean value μ and variance σ^2, i.e., the prediction step will project the current state and error covariance forward in time to obtain the *a priori* estimates for the next time step. The measurement update step then feeds back the measurement into the *a priori* estimate to obtain an *a posteriori* estimate [5].

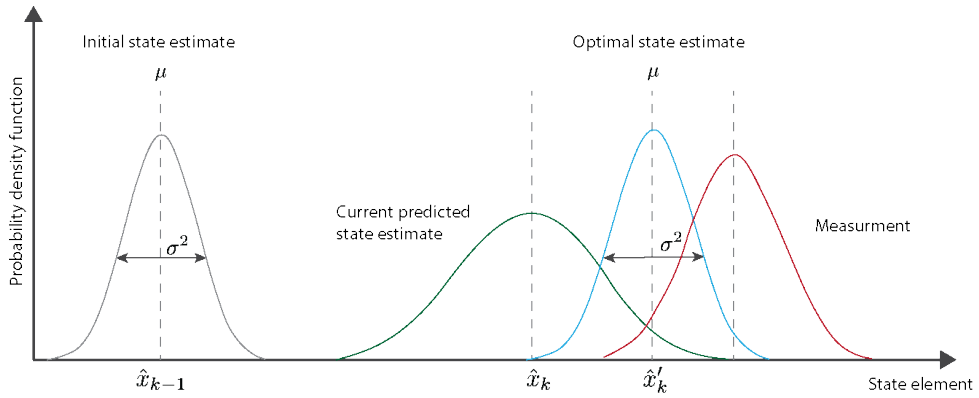


Figure 3.10: Probability density function of state estimates and measurements displaying how the measured state is combined with the predicted state estimate, $\hat{\boldsymbol{x}}_k$ in order to produce an optimal state estimate, $\hat{\boldsymbol{x}'}_k$.

We make this possible by taking each point in our original estimate and move it to a new predicted point, governed by the parameters that dictate how the system would behave if that original estimate was correct. The prediction step can be represented by a state transition matrix, $\boldsymbol{F}_k$:

$$\boldsymbol{F}_k = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \tag{3.46}$$

and:

$$\hat{\boldsymbol{x}}_k = \begin{cases} \boldsymbol{p}_k = \boldsymbol{p}_{k-1} + \Delta t \boldsymbol{v}_{k-1} \\ \boldsymbol{v}_k = \quad\quad\quad \boldsymbol{v}_{k-1} \end{cases} \tag{3.47}$$

This gives us the state estimate:

$$\hat{\boldsymbol{x}}_k = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \hat{\boldsymbol{x}}_{k-1} = \boldsymbol{F}_k \hat{\boldsymbol{x}}_{k-1} \tag{3.48}$$

So for each time we want to update the covariance matrix, since for a general case where $Cov(\boldsymbol{A}x) = \boldsymbol{A}\Sigma\boldsymbol{A}^T$, where Σ denote a covariance matrix, we get that:

$$\begin{aligned} \hat{\boldsymbol{x}}_k &= \boldsymbol{F}_k \hat{\boldsymbol{x}}_{k-1} \\ \boldsymbol{P}_k &= \boldsymbol{F}_k \boldsymbol{P}_{k-1} \boldsymbol{F}_{k-1} \end{aligned} \tag{3.49}$$

This however only holds true for a system where the velocity is constant and does not account for any acceleration the system may carry. So if we assume that we expect an acceleration, α, we get:

$$\hat{\boldsymbol{x}}_k = \begin{cases} \boldsymbol{p}_k = \boldsymbol{p}_{k-1} + \Delta t \boldsymbol{v}_{k-1} + \frac{1}{2}\alpha \Delta t^2 \\ \boldsymbol{v}_k = \qquad\qquad \boldsymbol{v}_{k-1} + \alpha \Delta t \end{cases} \tag{3.50}$$

Or in matrix form:

$$\hat{\boldsymbol{x}}_k = \boldsymbol{F}_k \hat{\boldsymbol{x}}_{k-1} + \begin{bmatrix} \Delta t^2 \\ \Delta t \end{bmatrix} \alpha = \boldsymbol{F}_k \hat{\boldsymbol{x}}_{k-1} + \boldsymbol{B}_k \boldsymbol{u}_k \tag{3.51}$$

, where $\boldsymbol{B}_k$ is the control matrix and $\boldsymbol{u}_k$ is the control vector used as input to the system.

The above equation (3.51) would hold for an ideal system without any external uncertainty, but real systems do in most cases operate in environments within the presence of unknown external forces, modelled as noise. To overcome this issue, we treat these external influences as noise with covariance, $\boldsymbol{Q}$, which we add after every prediction step:

$$\begin{aligned} \hat{\boldsymbol{x}}_k &= \boldsymbol{F}_k \hat{\boldsymbol{x}}_{k-1} + \boldsymbol{B}_k \boldsymbol{u}_k \\ \boldsymbol{P}_k &= \boldsymbol{F}_k \boldsymbol{P}_{k-1} \boldsymbol{F}_k^T + \boldsymbol{Q}_k \end{aligned} \tag{3.52}$$

So in summary, the optimal state estimate, $\hat{\boldsymbol{x}}_k$, is a prediction made from the previous optimal state estimate, $\hat{\boldsymbol{x}}_{k-1}$, with an added correction for the process noise Q which is assumed to be normal distributed. The uncertainty is predicted from the previous uncertainty with an addition of the environmental uncertainty. The final part of the algorithm is the *measurement* part. As we receive new data, we are storing it into a vector, $\boldsymbol{z}$ since we are often logging more than one parameter of the system. These measurements may contain some level of noise, here denoted as $\boldsymbol{R}$:

$$\boldsymbol{z}_k = \boldsymbol{H}\boldsymbol{x}_k + \boldsymbol{R}_k \tag{3.53}$$

The matrix, $\boldsymbol{H}$ in equation 3.53 above is a general matrix of which we multiply the state into to convert it into a measurement matrix, meaning that for e.g., the state $\boldsymbol{x}_k$ as in equation 3.45, we set $\boldsymbol{H} = \begin{bmatrix} 1 & 0 \end{bmatrix}$. So when we receive the measurement values, we are expecting that:

$$\hat{\boldsymbol{z}}_{k-1} = \boldsymbol{H}\hat{\boldsymbol{x}}_{k-1} \tag{3.54}$$

However, in many cases:

$$\boldsymbol{y} = \boldsymbol{z}_{k+1} - \hat{\boldsymbol{z}}_{k+1} \neq 0 \tag{3.55}$$

, meaning that there is often a difference, $\boldsymbol{y}$ (also commonly known as the *innovation*[38]) between the estimated value and the measured value. Ideally, this difference would be zero, but in order to calculate how much the state would change based on this difference, we incorporate it into the state estimation equation as:

$$\hat{\boldsymbol{x}}_{k+1} = \boldsymbol{F}\boldsymbol{x}_k + \boldsymbol{K}\boldsymbol{y} \tag{3.56}$$

, where $\boldsymbol{K}$ is known as the *Kalman Gain*, which is proportional to the ratio between the process noise and the noise in the measurement, i.e., it is the gain we should use to create the new optimal state estimate, $\hat{\boldsymbol{x}'}_k$[21]. The derivation of the Kalman Gain is quite extensive and will therefore not be covered in this thesis. The interested reader is referred to [27] and [38] which covers the Kalman filter and its derivation in great detail. This gives us the final equations for the update step:

$$\begin{aligned} \hat{\boldsymbol{x}'}_k &= \hat{\boldsymbol{x}}_k + \boldsymbol{K}'(\boldsymbol{z_k} - \boldsymbol{H}_k \hat{\boldsymbol{x}}_k) \\ \boldsymbol{P}'_k &= \boldsymbol{P}_k - \boldsymbol{K}' \boldsymbol{H}_k \boldsymbol{P}_k \end{aligned} \tag{3.57}$$

, where:

$$\boldsymbol{K}' = \boldsymbol{P}_k \boldsymbol{H}_k^T (\boldsymbol{H}_k \boldsymbol{P}_k \boldsymbol{H}_k^T + \boldsymbol{R}_k)^{-1} \tag{3.58}$$

Putting it all together, the prediction part of the algorithm projects the state ahead given the initial estimates and the parameters from the measurement update step. It also projects the error covariance to the time step ahead given the same parameters. The measurement update step computes the Kalman gain and updates the estimate with the measurement $\boldsymbol{z}_k$. It then updates the error covariance and feeds it back to the prediction step. This runs as an iterative process and can be run in real time for as many times as we like.

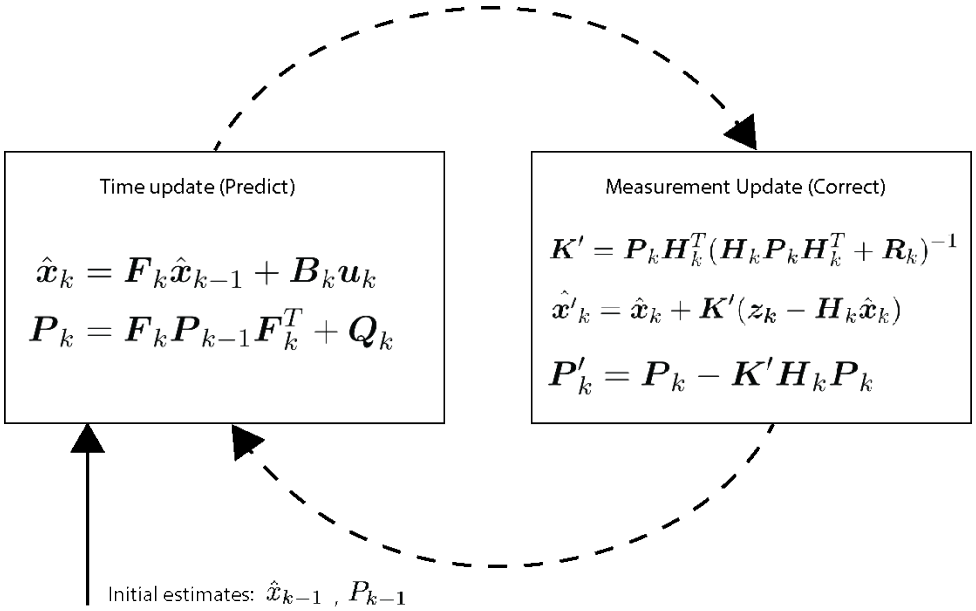


Figure 3.11: Diagram showing the basic operation of the Kalman filter, combining the equations from the prediction part (equation 3.52) and the measurement update part (equations 3.58 and 3.57).

So by using this idea of a prediction followed by a correction in order to determine the state of the system given some amount of uncertainties, the Kalman Filter can be used to model the behaviour of any linear system with good accuracy.

Chapter 4

Implementation

This chapter will cover the implementation of the various hardware- and software components and the communication between the computer and the MAV, as well as the general control structure of the whole system. We will also discuss in further detail the configuration of the camera, implementation of the Kalman filter and PID controller, as well how the detection algorithm and relative pose estimation was implemented in practice through OpenCV and the ArUco library. Lastly, we will go through the algorithm that connects all components together and allows the MAV to perform autonomous navigation and landing.

4.1 Added sensors

When using a mono- or stereo camera configuration intended for inferring information about the environment of any robotics system, two possible options come to mind; having the cameras mounted in a fixed position, often referred to as *eye-to-hand*, or in a mobile position where the cameras are attached to the robot itself, known as *eye-in-hand*. One could also consider a combination of the two with one camera mounted in a fixed position external to the MAV and having one camera attached to the vehicle. In this project, we utilize a mono-camera configuration for the marker detection and relative pose estimation because of the low weight, low price and simplicity in calibration it entails.

The RGB camera is mounted in an eye-in-hand configuration as seen in figure 4.1a below, together with the optical flow deck which is mounted to the bottom of the unit. The optical flow sensor works as a complement for stabilizing the craft by tracking features (as described in section 3.4.1) in the X-Y plane and their motions between frames. Thus, the overall control structure consists of two control layers, an inner loop implemented onboard in the firmware of the Crazyflie 2.0 that provides stabilization of the craft with input from the IMU and the optical flow deck, and an outer control loop that provides stabilization of the MAV in terms of position and orientation with respect to the marker. The input to the outer control loop is determined by the relative translations and rotations computed from the images taken by the camera mounted on top of the MAV.

(a) Monocular eye-in-hand camera configuration.

(b) Optical flow deck mounted to the bottom of the MAV

Figure 4.1: RGB-camera and optical flow deck mounted on the Crazyflie 2.0

This proves to be a very reliable combination because of the two sensors has its own designated purpose, with the RGB camera being responsible for acquiring the data used for computation of the relative position and orientation of the quadcopter with respect to the marker, and the optical flow deck being responsible for maintaining stability of the craft itself whilst in flight. This means that the stability of the vehicle does not get compromised when the marker is not detected by the RGB-camera. Note however that solutions using only the RGB-camera could in theory be possible, as presented by [6] and [11], but may require a different approach than the one presented in this thesis since the detection of the marker and pose estimation requires a heavy computational cost.

4.2 The Crazyflie Python API

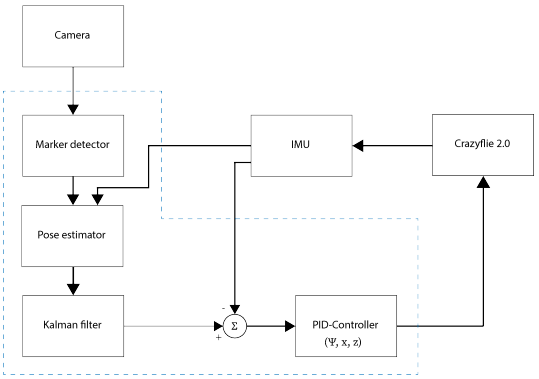
In order to easily control the Crazyflie 2.0 from the computer, an API(Application Programming Interface) for Python provided by Bitraze AB was used, giving high-level access to a communication protocol for logging of variables and transmission of control commands to the MAV. The computer transmits and receives data from the MAV by utilizing the Crazyradio PA which handles communication through UART (2.4GHz) and can be plugged directly into a free USB port on the computer. The received data contains information read by the on-board sensors of the quadcopter, which in the case if this project is the current voltage of the battery and the rotational movement along each axis of the body fixed frame of the MAV, determined by the IMU of the quadcopter. The library is asynchronous and is based on callbacks for events, meaning that each time we receive data, it is required to match the time of reception with the time of computation. This is handled through sending a time-stamp with each transmission from the MAV which can then be synchronized with the internal clock of the computer.

However, before we can begin to send and receive data between the quadcopter and the computer, we must initiate a link between the computer and the Crazyflie 2.0 that we want to connect to. The API allows us to scan for available interfaces and opens up a communication link between the Crazyradio PA and the desired Crazyflie 2.0. Once we have the established the communication link, a log configuration must be written to the firmware of the Crazyflie 2.0. This configuration contains information about what variables we are interested in receiving in real time from the MAV and at which period they should be logged. Each log-packet is limited to a maximum of 32 bytes, giving us access to about 14 different variables. This is more than sufficient for the purpose of this project, since we are primarily interested in the roll(Φ), pitch(Θ), yaw(Ψ) and the current battery level. When the configuration is set up and validated, it is written to the firmware and sends back the data to the computer each time we initiate the callback through the logging framework.

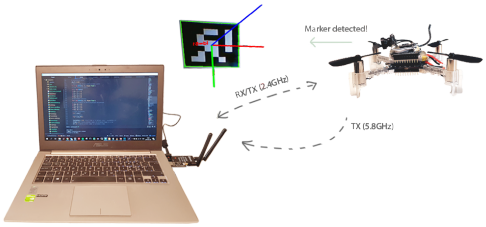
4.3 General control structure

After having discussed the principles of obtaining data from the different sensors, it's worth outlining the general architecture of the system. In summary, the computer acting as the ground station receives the images from the RGB-camera in the form of a continuous video feed through analogue transmission over the 5.8GHz band and it can perform a two-way communication with the Crazyflie 2.0 through the Crazyradio PA, logging data and sending control signals to the vehicle. Through the ArUco library, each obtained image frame is processed in the computer and once a marker has been detected by the algorithm, the pose of the quadcopter relative to that marker in that image image frame is calculated and is fed to a Kalman filter. The Kalman filter makes an optimal estimation of the relative pose of the MAV, given the previous state data, the measurement data and the error covariance matrix. An error signal is then computed and the PID controllers output the commanded control signal to the Crazyflie 2.0 in order for the MAV to align and position itself towards the marker.

The control signals from the PID controllers is comprised of the relative yaw(Ψ)-angle and translation in x- and z-direction, x defining the horizontal alignment offset of the quadrotor in the world frame and z defining the distance between the detected marker and the quadrotor in the world frame in centimetres. The translation in y-direction is intentionally left out because we do not want the quadcopter to fall to the same height as the marker before we have initiated the landing procedure. Figure 4.2 below gives a basic idea of the overall control structure with some simplifications.



(a) Block diagram displaying the simplified control structure of the system. The components contained within the blue dotted line define operations that occur in the computer external to the MAV



(b) Physical representation of the system architecture

Figure 4.2: Overall system architecture

Ideally, all computations should be done on-board, eliminating the need of an external computer which will introduce a time delay in the system due to e.g., time loss in transmission and processing. However, having all computations made on-board may in the case of this platform require extensive hardware modifications because of its low weight and limited computation power. Therefore, it was decided to have an external computer do all the heavy computations.

4.4 Camera calibration using ArUco

Before we can begin using the ArUco library for marker detection and pose estimation, the camera must be calibrated in order to be able to recover information about the size of the markers in world units and determine the pose of the camera in the scene with good accuracy. The ArUco module provides this option through OpenCV from which we can obtain the camera intrinsic parameters and the distortion coefficients. The procedure of calibrating the camera needs only to be done once since the properties of the camera remain unchanged unless we change its physical configuration by e.g., replacing the lens.

Conventionally, the camera is calibrated by providing multiple images of a calibration pattern. In many applications, the calibration pattern comes in the form of a checkerboard with known 3D world points from which the corresponding 2D image points are calculated. Using this 3D to 2D correspondance, it is possible to solve for the camera parameters. This same approach is used in this project but with the difference being that the calibration is performed based on finding the corners of multiple ArUco markers set up in a grid. The ArUco module also provide a second option where the ArUco markers are set up in a checkerboard pattern, known as a *ChArUco Board*. The benefit of using ChArUco or the ArUco board rather than the traditional checkerboard patters is that they allow for partial views and occlusions in the image, meaning that not all corners need to be visible in all the viewpoints. OpenCV recommends using the ChArUco corners approach[22] because of its higher accuracy, however, the ArUco approach has proved to be sufficient for the scope of this project.

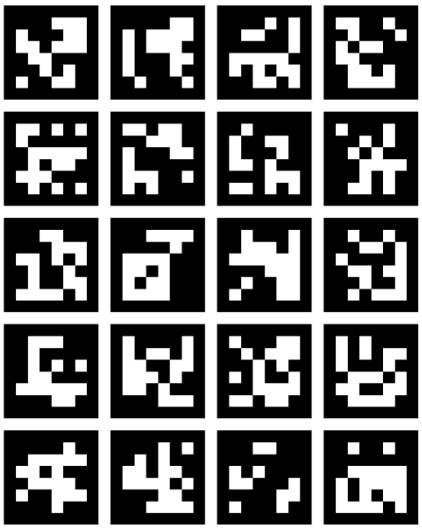


Figure 4.3: The *ArUco Board* used during calibration. It has five rows and four columns and the identifiers of each marker in the board ranges from 0 to 19.

The ArUco board may also be used for post calibration pose estimation since the obtained pose from a detected board of markers is usually more accurate than of a single marker. This is because when using a set of independent markers, the pose of each marker can be individually estimated, providing a higher amount of 3D to 2D point correspondences. This also implies that partial occlusions of the board can take place without significantly impacting the performance of the pose estimation algorithm. However, because of the low resolution of the camera used in this project, using an ArUco board for pose estimation during flight is not feasible. The board would have to be very large in order for the vision system to properly detect and validate the markers at a larger distance, for which reason, the decision fell on detection of single markers during flight.

4.5 Marker creation and detection using the ArUco module

As previously mentioned in section 3.4, the ArUco library allows for a selection amongst a set of hardcoded dictionaries, namely the *ARUCO_MIP_25h7*, *ARUCO_MIP_16h3*, and *ARUCO_MIP_36h12*, but also allows for custom dictionaries. When choosing a dictionary, one should consider that the number of bits contained in the marker will govern the distance of which it may be detected, e.g., a marker with an internal binary matrix of size 4×4 will be less difficult and less prone to erroneous detection at a larger distance, than say a marker with an internal matrix of 6×6. During development of the ArUco library, it was concluded that the *ARUCO_MIP_36h12(6x6)* was the optimal dictionary considering the trade-off between robustness and size [25], which is the reason this specific dictionary was employed in this project.

When generating the marker, we should also consider the quantity of markers that we want to be able to detect. The *ARUCO_MIP_36h12* is a 36 bit dictionary and can be set to hold a maximum of one thousand valid markers, which is more than sufficient for this project. Lastly, we should define the size of the marker image before we are ready to print it and place it in the environment around the MAV. In this case, the marker size was set to 200×200 millimetres. Knowing the physical dimensions of the marker is very important because an inaccurate 3D to 2D correspondance will yield an inaccurate pose estimation of the camera. After having defined and printed the markers, the side length of the markers were measured and we are ready to begin implementing ArUco library into the code.

Given that we have calibrated the camera such that we have obtained information about the $\boldsymbol{K}$-matrix that holds the elements describing the focal length of the camera, its optical center and the slant factor (see section 3.3.3), and information about the camera distortion coefficients (see section 3.3.4), the image taken by the camera can be rectified and interpolated in order to remove any distortion effects in the image before it is given as input to the detection algorithm. The detector also requires that the input image is first converted to a grayscale representation of the original color image. The detector function will then return a list of all corners found for each detected marker, its respective identifier and an array containing information about the rejected image points corresponding to detected squares in the image that do not present a valid codification.

4.6 Relative pose estimation

When the four corners of the marker has been detected and has been validated with the identifier that we want to track, the pose of the camera can be estimated through a homography transformation function. The function returns the transformation as two vectors, one vector holding elements describing the translation between the reference frame of the marker and the camera reference frame, and one vector holding the elements describing the relative rotation between the two. The function relies on the theory discussed in section 3.3.3 and 3.4 and the problem of pose estimation is often referred to as the *Perspective-n-Point* or PNP -problem in computer vision. OpenCV has implemented this as a method called *solvePNP()* in Python which provides several algorithms to solve for the relative pose by *Direct linear transformation*(DLT) or by the *Levenberg-Marquard optimization*-method[35].

Once we have obtained the rotation and translation vectors, the next step is to determine the rotation matrix, $\boldsymbol{R}_{zyx}$, which holds the rotation between the two reference frames in a similar fashion as seen in equation 3.12. By using *Rodrigues formula* (equation 3.13) , the rotation matrix can be calculated and its respective Euler angles can be computed through following the procedure as noted in equations 3.17 through 3.19.

4.7 Signal filtering

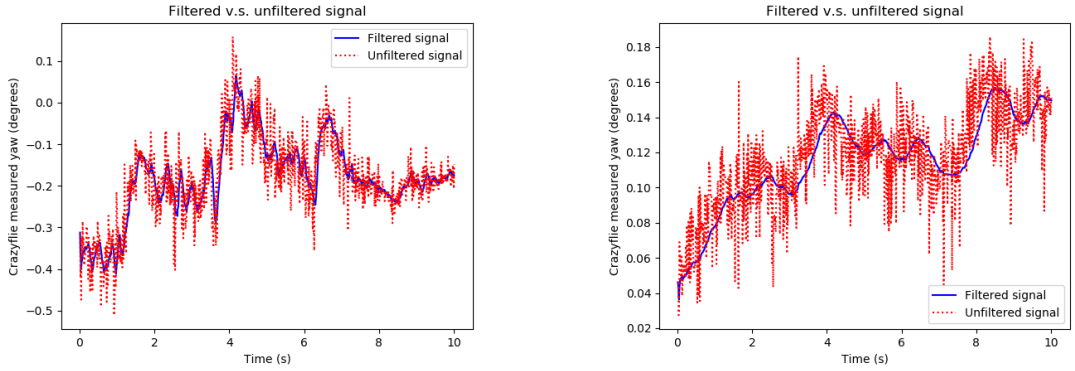
Measurements taken from real world systems do often contain some level of noise due to e.g., vibrations in the system or electromagnetic interference (EMI). In order to reduce the unwanted features in the signals that we receive from the IMU and the camera sensor, we utilize two types of filters in this thesis; A *Moving Average Filter* and a Kalman filter. Note that the latter is not only used for filtering out noise present in the signal, but also for providing estimations about unknown parameters when given incomplete or corrupt detection of the target marker.

4.7.1 Moving Average Filter

The moving average filter works by simply taking m number of samples of the streaming data and averaging them over a period of time. The moving average filter can be thought of as a special case of the regular *FIR*(Finite Impulse Response)-filter and computes the average of the input signal with a finite sliding window:

$$MovingAvg = \frac{x[n] + x[n-1] + ... + x[n-m]}{m+1} \tag{4.1}$$

This allows us to smooth out the noise in the streamed sensor data, as can be seen in figure 4.4 below. Note however that the size of the sliding window, m must be defined carefully as very large values of m will yield a very slowly decaying or increasing filtered output which may decrease the performance of the overall system.



(a) Window size: $m = 10$.

(b) Window size: $m = 60$.

Figure 4.4: An example of the moving average filter in action with different defined window sizes

In this example, the Crazyflie 2.0 was placed in a stationary position and the rotation about its z-axis was measured by the IMU for ten seconds. The data was streamed in real-time to the computer and logged in memory. As is obvious upon inspection of figure 4.4, by using a small window size, the filtered output signal follows the raw data signal through its peaks and valleys, but increasing the window size yields a smoothed output signal by suppressing sudden changes in the signal.

4.7.2 Implementation of Kalman filter

The data that we are looking to estimate and filter in the case of this project is the *relative* yaw angle (Ψ) and the translation ${}^{\mathcal{C}}\mathbf{t}_{\mathcal{M}}$ between the body fixed frame of the quadrotor and the reference frame of the detected marker, meaning that we need to define the state vector holding the elements: ${}^{\mathcal{C}}\mathbf{x}_{\mathcal{M}} = \{\Psi, \dot{\Psi}, t_x, \dot{t_x}, t_z, \dot{t_z}, t_y, \dot{t_y}\}$, where the leading superscript $\mathcal{C}$ is the

camera reference frame which is assumed to have its axes lie collinear and with zero translation in relation to the reference frame of the MAV for simplification. Subscript $\mathcal{M}$ denote the reference frame of the marker. Having a state vector containing eight elements imply that we need a state transition vector initially defined as:

$$\boldsymbol{F}_k = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

, where each non-zero element above the diagonal in each column of the matrix defines the time Δt between the states. The measurement matrix, $\boldsymbol{H}$ is then initiated as:

$$\boldsymbol{H} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{4.3}$$

, where a non-zero value represent the elements of which we want to measure and estimate. Thereafter, we define the uncertainty matrix, $\boldsymbol{R} = k_1 \boldsymbol{I}_{n \times n}$, where k_1 is the uncertainty factor and n is the number of parameters that we want to estimate, in this case four. Lastly, we define the process noise matrix, $\boldsymbol{Q} = k_2 \boldsymbol{I}_{2n \times 2n}$, where k_2 is a constant determining the magnitude of the process noise.

Each measurement we make is given as input the Kalman filter together with the previous estimated state and the calculated error covariance matrix. This is an iterative process and the Kalman filter gets updated with a new state estimate for each iteration. However, having the filter defined in this way implies that we are expecting a constant velocity, meaning that we have not accounted for any acceleration. This means that if the detected marker exhibit rapid movement in the image frame and detection is suddenly lost for an amount of frames, since the control signal to the MAV relies on the estimated data given by the Kalman filter, it would follow a linear motion proportional to the estimated velocity. One way to solve this would be to instead implement an extended Kalman filter (EKF), however, I opted for a different approach where if the desired marker has not been detected for n number of frames in a row, the velocity for each state element is exponentially decreased such that for each time step, $k_n \leq n_{max}$, where k_n denote a time step with an undetected marker:

$$\hat{\boldsymbol{x}'}_{k_n}(\boldsymbol{v}_k) = \hat{\boldsymbol{x}'}_{k_n}(\boldsymbol{v}_{k-1})\alpha \tag{4.4}$$

, and α denote the diminishing factor.

4.8 PID control

As mentioned in section 4.1, two main controllers are used in this system, one that runs internally in the firmware of the Crazyflie 2.0 and one set that runs in the computer that sends the commanded trajectory to the MAV, determined by the computed relative pose estimation. The set of PID controllers that run externally to the MAV has three different command variables, the relative yaw angle and translation in x- and z direction, hence, three PID controllers are implemented in the software. These controllers continuously compute the error signals as the difference between the desired setpoint and the measured value of the current pose of the quadcopter relative to the marker and thereafter applies a correction based on the proportional, integral and derivative terms for each controller.

4.8.1 Tuning the PID controllers

In order to have the system operate in a stable and desirable way, each controller must be tuned by defining values for the proportional gains, K_p, K_i and K_d. This was achieved by evaluating the step response for each of the three controllers through experiments. The AMIGO method was chosen as the primary tuning method as it has been proven to be a reliable method for tuning the PID controllers of the Crazyflie 2.0 in previous work by [16].

Each controller was tuned with the AMIGO method separately by turning off the other controllers whilst focusing on one controller at the time. After having obtained values of the dead time, L, the time constant, $T_{63\%}$ and the static gain, K, the gains were calculated by following the procedure described in section 3.5.1. However, this gives us only an indication of feasible values of the respective gains and requires some additional manual tuning. By evaluating the step response in terms of rise time, overshoot, settling time and steady state error, following table 4.1 below gives us a proposal of how to continue tuning each parameter manually:

Table 4.1: Effects of independent tuning by increasing the gains, K_p, K_i and K_d, in credit to [18].

Gain	**Rise time**	**Overshoot**	**Settling time**	**Steady-State Error**	**Stability**
K_p	decrease	increase	small change	decrease	degrade
K_i	decrease	increase	increase	eliminate	degrade
K_d	increase	decrease	decrease	-	Improve for small values of K_d

For safety reasons, a limit to a maximum velocity for the MAV was set. All experiments were conducted in a very small area, meaning that having the quadcopter move rapidly in either direction could induce a hazard to both the drone itself and its surroundings. The translational velocity limitation was set to $8 cm/second$ and the angular velocity was limited to about $90 degrees/second$, which was taken into consideration when performing the manual tuning.

4.9 Application

Once the MAV is in flight and has detected and validated the marker in its surroundings through information captured by the on-board camera, command signals will be sent to the vehicle such that it orient itself towards the target and it will stay locked on the target for a certain time before it begins to approach the target, i.e., a commanded signal is sent from the computer to the quadcopter containing the translation in the cameras z-axis (see figure 3.4) relative the detected marker. The time the quadcopter stays locked on target and in hover can easily be changed by modifying the target lock-time defined as a constant in the source code. The landing process involve a linear motion towards the marker with constant velocity and when the MAV has reached within a defined vicinity of the marker, it stops the linear motion and slowly descends until it turns of all four motors. This whole process can be seen as illustrated in figure 4.5 below.

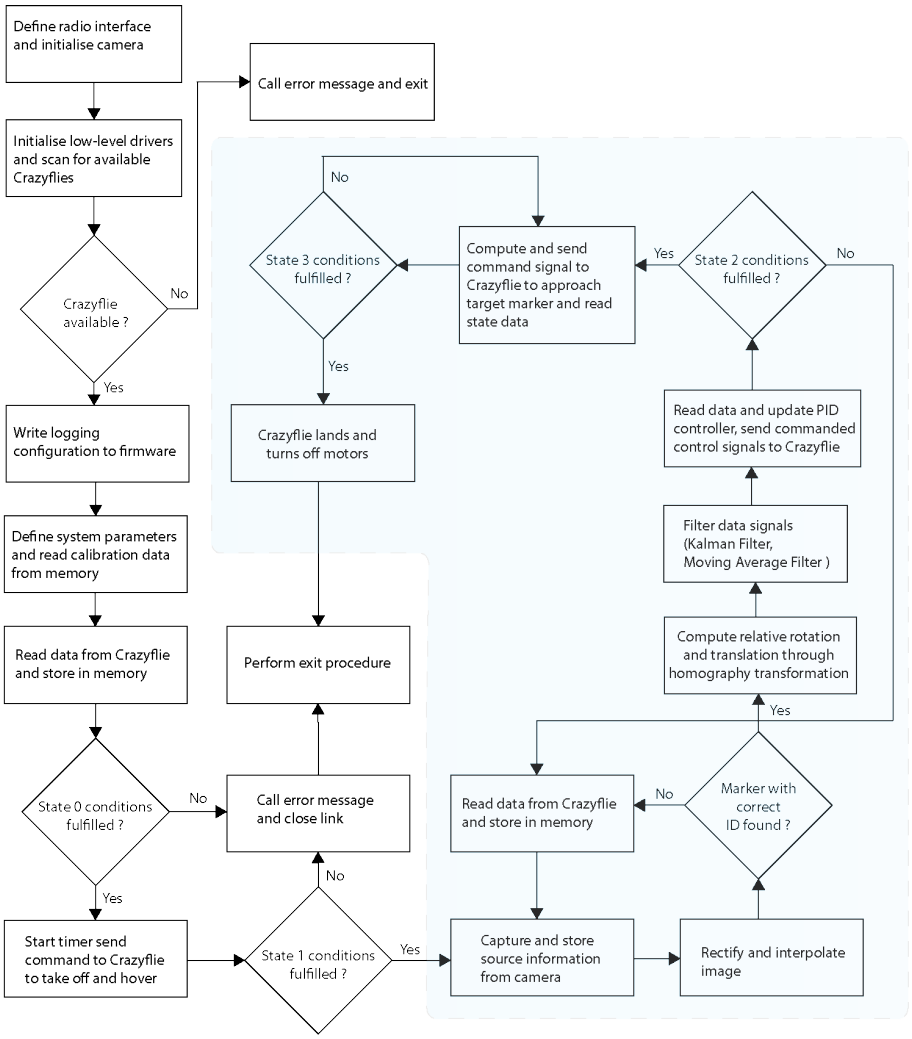


Figure 4.5: Flow diagram of the application

The main loop of the system is contained within the blue section in figure 4.5 and the initialization step of the whole process lies outside the blue section. The whole process is comprised of four main state conditions that have to be fulfilled in order to move on to the next stage in the process. The first state takes into account that the battery level is above a certain threshold and that the link quality between the computer and the quadcopter is sufficient. If every condition is fulfilled, a signal is sent to the Crazyflie 2.0, commanding it to

take off and hover at a predefined height. Given that the quadcopter remains stable after take off and is capable of receiving command signals and transmitting data back to the computer, we check that the images received by the on-board camera are not corrupt and we proceed into the main loop. The first step in the main loop captures the information received by the on-board camera sensor and are transmitted through wireless communication to the computer where the images get rectified and put through the ArUco marker detector function. This function also validates the detected marker identification and computes the relative rotation and translation between the marker reference frame.

The output data from the detector function is then filtered through a Kalman filter which also estimates the relative pose of the camera if the detection gets lost for a number of frames, as described in section 4.7.2. The PID control signals are then computed and sent to the MAV and the process repeats itself until the vehicle has locked in its position and attitude towards the marker and has remain locked on target for a given time. When these conditions have been fulfilled, the quadcopter is commanded to move towards the marker until it has reached within a certain distance towards the marker. Finally, as this distance has been reached, it is commanded to land and turn off power to all four motors.

Chapter 5

Results and evaluation

Ths chapter will cover the results of the experiments conducted during development of the vision system. We will also discuss the outcome from each experiment and evaluate the results, as well as discuss some of the underlying causes of the outcome and possible ways of improving the results.

5.1 Camera calibration

Through following the process as described in section 4.4, the camera intrinsic parameters and distortion coefficients were estimated by providing 85 different images of the ArUco board taken from various angles to the calibration function. The function detects the four corners around each marker in the board and estimates the camera parameters from which each image can be undistorted and we are given a set of corresponding translation and rotation vectors from each image. This allows us to re-project the corner points in the image and we can evaluate the performance of the calibration through measuring the average re-projection error. The re-projection error gives us a good approximation of the accuracy of the calibration and is defined as the distance between the detected corner keypoint in the image and the corresponding point in the world frame, projected in the same image. A very high average re-projection error indicate that the confidence between points in the image frame and the world frame is very low, meaning that the translation and rotation between the two reference frame cannot be computed with high certainty. Therefore, during calibration, we want the re-projection error to be as low as possible. Figure 5.1 below shows the re-projection error after calibrating.

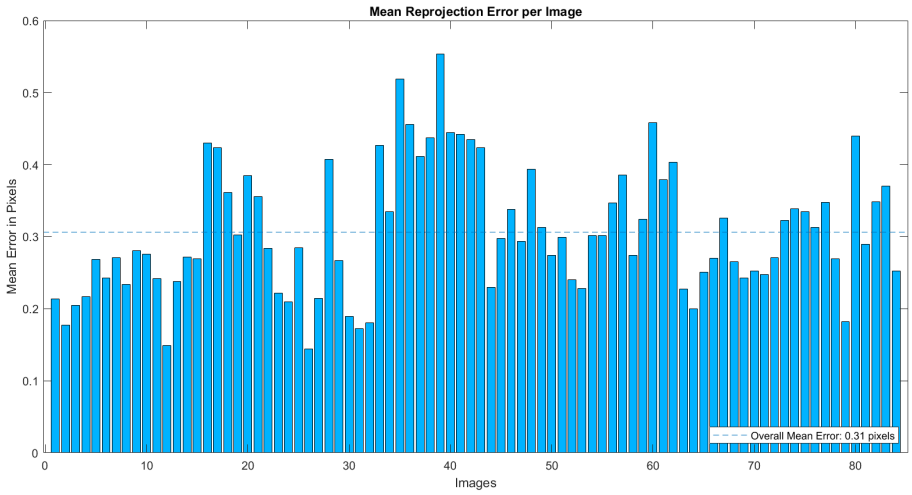


Figure 5.1: Re-projection error given a sample of 85 images

By looking at the calibration data, we see that the overall mean re-projection is only 0.3062 pixels, which can be considered quite good and is more than acceptable for the purposes of this project. Using 85 sample images as input to the calibration function may be considered an exaggeration and removing some of the samples that exhibit a large re-projection error would lower this value significantly because providing multiple images as input to the calibration function entails in the function computing the $\boldsymbol{K}$-matrix for all combinations of those images and the error is the mean between all those computed matrices. Next, it is of interest to examine the uncertainty of each estimated parameter. By evaluating the standard deviation σ of each estimation, we can get an idea of how accurately the camera parameters and subsequently the frame transformation vectors could be estimated and thereby obtain a concept of how much confidence we can have in the parameters conforming to reality. Table 5.1 below shows the elements contained in the computed camera matrix and the standard deviation of each estimated intrinsic parameter.

Table 5.1: Results of camera calibration

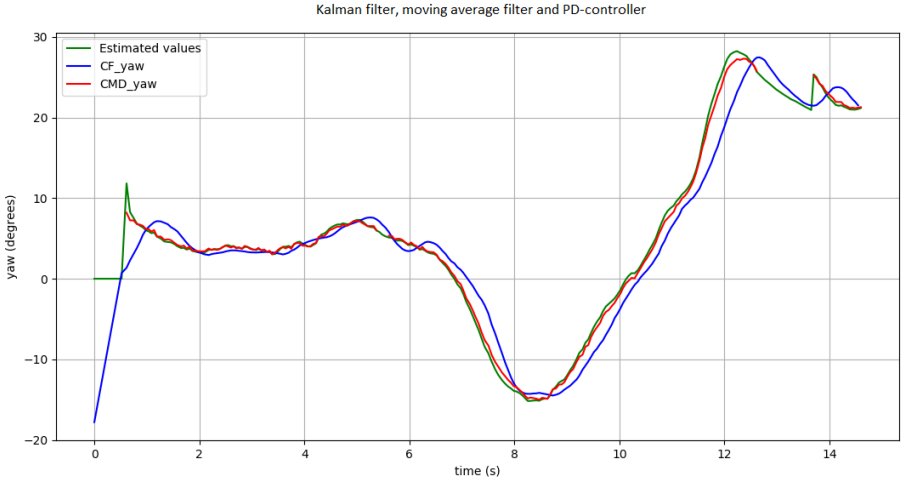
Camera intrinsics from calibration		
Parameter	Value (pixels)	σ (pixels)
Focal length (f_x, f_y)	[248.1442, 244.0496]	$\pm$ [0.2506, 0.2445]
Principal point (c_x, c_y)	[308.4223, 2.551080]	$\pm$ [0.1191, 0.1459]
Skew factor (s)	-0.2630	$\pm$ 0.0403
Radial distortion	[0.2141, -0.2542, -0.0708]	$\pm$ [0.0014, 0.0022, 0.0010]
Tangential distortion	[-0.0035, 0.0015]	$\pm$ [0.0002, 0.0002]

5.2 Filtering and estimation

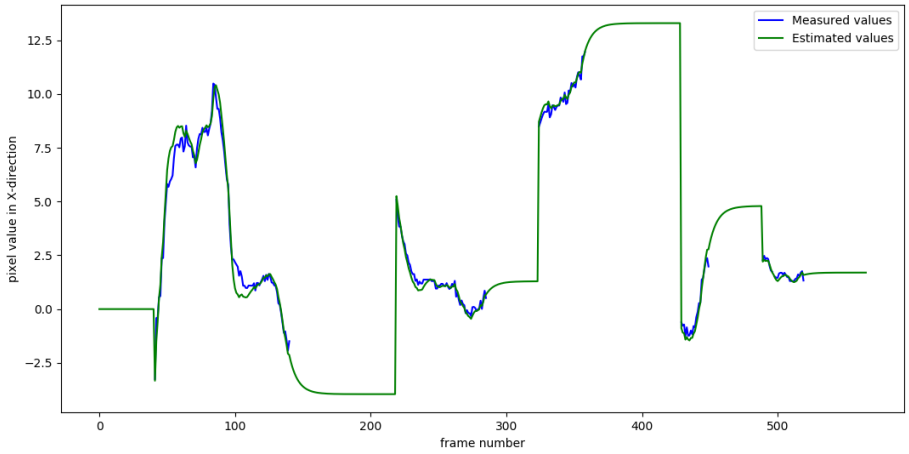
When implementing the Kalman filter, it was necessary to perform some initial experiments in order to determine how it should be utilized and what parameters to set. Firstly, the filter should be specified with the number of dimensions contained in the state vector, i.e., how many variables we want to filter and estimate. In the case of this project, we need to keep track of the relative yaw angle and translation in x,y and z-direction and their respective derivatives, meaning that we need eight elements contained in the state vector along with the 4×4 measurement matrix, $\boldsymbol{H}$. We also need to define the state uncertainty, $\boldsymbol{R}$ and the process

noise, $\boldsymbol{Q}$. A higher uncertainty factor will yield an output from the Kalman filter that deviates more from the measured values by giving the internal properties of the filter a higher priority, whilst lower values will make the filter output follow the measured values more accurately.

Figure 5.2a shows a plot over a short tracking sequence where the marker was moved in various angles in front of the MAV whilst it was hovering and was given attitude control signals such that it tries to always have its y-axis pointing collinear with the z-axis of the reference frame of the marker, radiating out from its center. The green line in figure 5.2a displays the estimated values given as output from the Kalman filter, the red line is the raw data output from the ArUco detection and pose estimation function and the blue line is the measured relative yaw angle of the quadcopter. As can be seen upon inspection of figure 5.2, at time $t \approx 5.5s$, the Kalman filter is able to make a good estimate of the movement of the marker when the measurement input data is lost. Also, at $t \approx 12.5s$, the detection of the marker is lost for approximately one second and the Kalman filter predicts the movement of the marker given the velocity at this time which gets reduced by the factor α for each iteration until the maximum number of frames, n_{max} has been reached as described in section 4.7.2.



(a) Experiment 1, yaw (Ψ): The red line displays the computed relative yaw angle, the blue shows the measured yaw angle of the vehicle and the green line is the estimated yaw angle which is transmitted to the MAV.



(b) Experiment 2, marker center (x-direction): The blue line displays the computed values and the green line displays the output data from the Kalman filter.

Figure 5.2: Kalman filter experiments

Similarly, in figure 5.2b, the Kalman filter was used in an experiment for tracking the centerpoint of the ArUco marker. In this experiment, the camera view was obscured at different points in time (sections where the blue line in figure 5.2b is absent) and moved to a new location. The filter was limited to output only an estimate for a maximum of 25 frames in a row when the marker was not detected and data was sampled only when the marker was detected or estimated by the Kalman filter, hence the peaks in the green line. Similar to the plot in figure 5.2a, the factor $\alpha = 0.85$ was set for diminishing the velocity $\hat{\boldsymbol{v}}_k = \alpha\hat{\boldsymbol{v}}_{k-1}$ for each iteration where an estimation has been made in the previous time step for a maximum n_{max} number of frames.

From these experiments, it was concluded that the parameter n_{max} was set to high in order for the MAV not to drift away when the marker exhibits rapid movements in the image frame. In the following experiment (see figure 5.3), the maximum number of frames, n_{max} was set to $n_{max} = 8$ for both estimation of the relative yaw angle and translation in x-direction. The marker was moved in various directions and orientations in front of the vehicle and control

signals was sent to the MAV such that it orients itself towards the marker, just as in the previous experiments, but this time with more aggressive displacements.

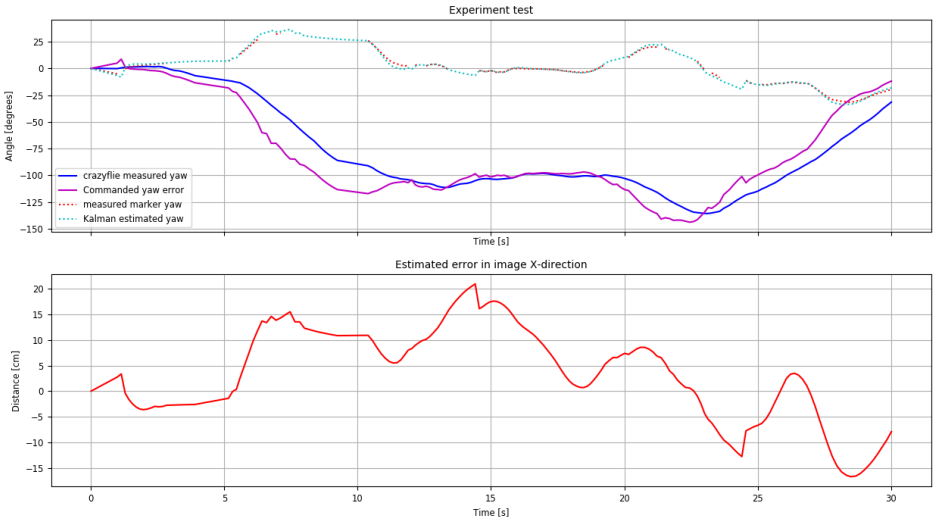


Figure 5.3: Experiment with more aggressive displacements of the marker. The blue line is the measured yaw angle of the MAV, the line in magenta displays the commanded signal computed through the PID controller, the dotted lines represent the estimated and measured error signal, $e(t) = \Psi_{cmd} - \Psi_{CF}$, where subscript CF denotes the Crazyflie 2.0 and cmd denotes the control signal.

Note that by displacing the marker more aggressively, the MAV takes longer time to reach the desired setpoint. The performance of the system in terms of the components listed in table 4.1 can be improved by proper tuning of the respective gains in the PID controllers. In the case of this experiment, the K_p, K_i and K_d gains in the outer controllers were tuned manually and by using the AMIGO-method presented in 3.5.1 The results from the tuning process will be discussed in the following section (5.3).

5.3 PID Tuning

The previously mentioned AMIGO- and Ziegler Nichols tuning methods can be performed either by intentionally putting the state where it presents oscillations around a given setpoint, or by evaluating the output response from the system when when given a defined control input. Due to the nature of the platform that is to be regulated, it was decided to use the step response for assessing the gains. Bringing the system into an oscillating state may be an unsafe option because the regulation is not a linearly independent process, meaning that instability along any of the axes of its body fixed frame may influence the stability of any other axis in that same frame.

First, the controller for regulating the relative yaw-angle of the quadcopter was tuned. Figure 5.4 below shows the step response when having the MAV placed in a position where the control signal: $\Psi_{cmd} = 45°$ and at a distance of 75 centimetres away from the marker along its z-axis. The I- and D terms in the PID controller were turned off by setting the values: $K_i = 0$, $K_d = 0$ and letting the proportional gain, $K_p = 1.0$.

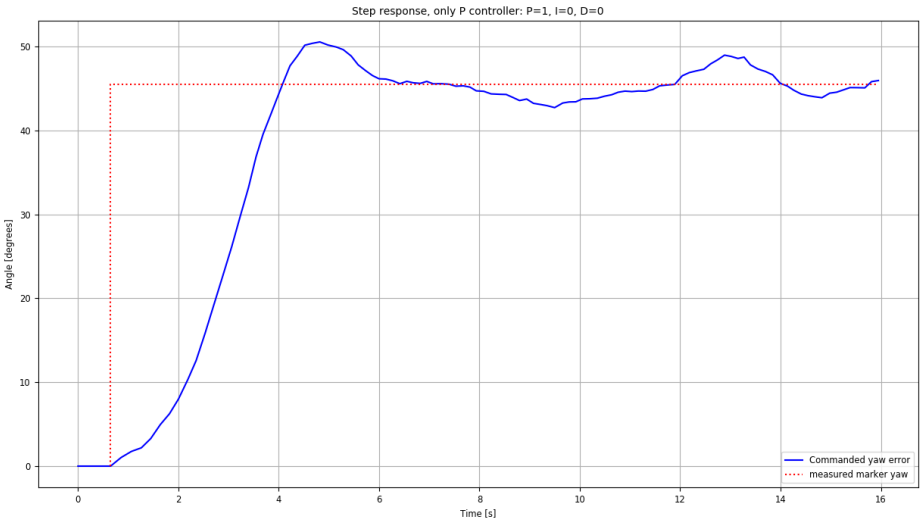


Figure 5.4: Step response (Ψ)

By analysing the step response, we can calculate the parameters $T_{63\%}$, L and the static gain, K. Since the step response exhibits oscillations around the setpoint, the static gain was given the value $K = 1$ for simplification and the the parameters L and $T_{63\%}$ were given the values shown in table 5.2 below:

Table 5.2: KLT parameters

$K = \frac{\Delta y}{\Delta u}$	$T_{63\%}$	L
1.0	2.44	0.6

The respective gains were then computed according to the equations given in table 3.2. This however did not improve the response in any way, but made the system quite unstable with significant overshoot and oscillations around the setpoint, as can be seen in figure 5.5 below.

Therefore, I opted for tuning the PID-controller for the yaw angle manually as described in section 4.8.1. The results when using the gains calculated through the AMIGO-method can be seen in figure 5.5 below.

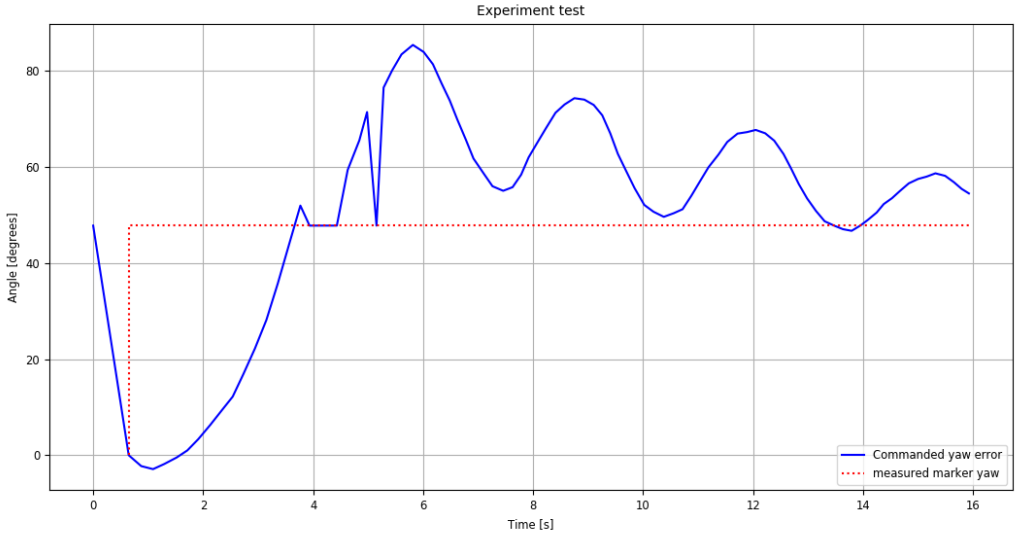


Figure 5.5: Step response for the relative yaw-angle when given a control input of $\Psi_{cmd} \approx 45°$, calculated by the ArUco marker detection and pose estimation algorithm.

Upon inspection of the step response in figure 5.5, it can be observed that after the output having initially breached the setpoint marked as a dotted red line in the plot, the output gets distorted and peaks down towards the setpoint before continuing oscillating. This occurs due to measurement errors and noise present in the signal which makes the graph look distorted and should not be considered when evaluating the overall performance of the controller, but should be handled by the noise suppressing filters. Because tuning this particular controller with the AMIGO-method does not prove to be reliable, it was decided to continue the tuning process through manual adjustments.

Manual tuning was performed by initially setting the proportional gain to $K_p = 1$ and the rest to zero, thereafter tuning the gains and evaluating the step response through looking at parameters such as the overshoot, settling time and rise time which each contribute to the overall stability. By following this process, I found that defining the gains as: $K_p = 1.0$, $K_i = 0.10$, $K_d = 0.045$ proved to give satisfactory results, although, using a standalone P- or PD controller could arguably be sufficient for the case of this project because the on-board attitude and velocity controller compensates for any biasing caused by e.g., imbalance in the platform or externally acting forces thanks to the optical flow deck.

Next, the regulator used for controlling the position of the MAV with respect to the marker in x-direction must be tuned. The tuning process was done by following a similar procedure as for the yaw-angle controller. The quadcopter was placed approximately 20 centimetres away in x-direction and 75 centimetres away from the marker in z-direction (x,z in the camera reference frame) and was then given control signals to orient itself by regulating its roll-angle such that the marker lies in the center of the x-axis in the image frame. The initial step response with the gains defined as: $K_p = 1.0$, $K_i = 0$ and $K_d = 0$ can be seen in figure 5.6 below.

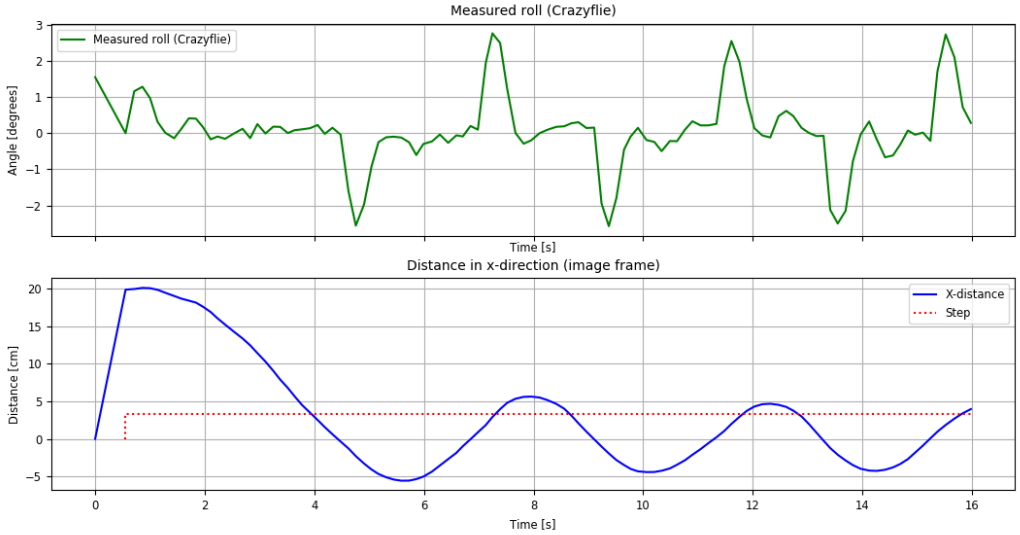


Figure 5.6: Step response

After estimating the parameters, $L, T_{63\%}$ and K, tuning with AMIGO yields results as shown in figure 5.7. The quadcopter regulates its translation in x-direction by controlling its roll-angle, hence the inclusion of the graph displaying the measured roll-angle.

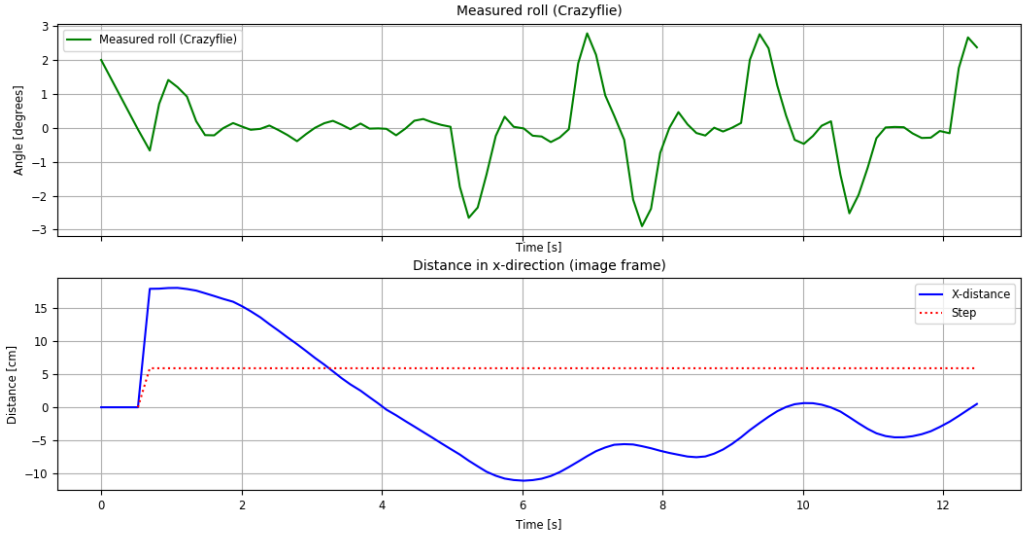
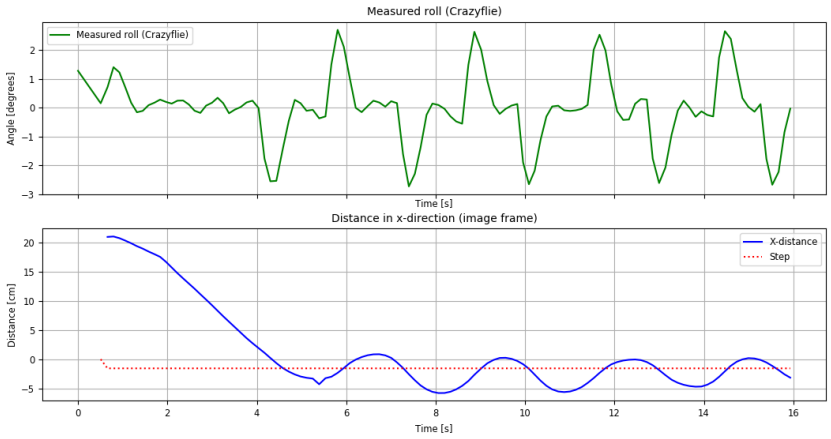


Figure 5.7: Step response

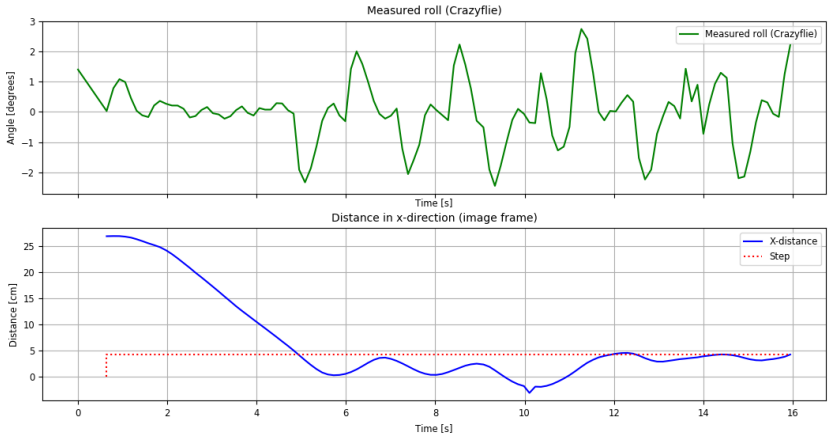
From this result, it is obvious that the controller requires some further manual tuning. By simply gradually decreasing the proportional gain and further tune the integral- and derivative gains, the system response became considerably more stable but still exhibits oscillations around the setpoint. As a countermeasure for reducing these oscillations, instead of letting the proportional term be determined by the error, $e(t)$, the PID controller was modified such that the P-term is fed the current input value of the PID controller:

$$u(t) = K\left(\beta e(t) - (1-\beta)x + \frac{1}{T_i}\int e(t)dt + T_d\frac{d}{dt}e(t)\right) \tag{5.1}$$

, where x is the measurement, $\beta = 0$ defines the *proportional-on-measurement* (PonM) and $\beta = 1$ defines the conventional way of computing the proportional on error. Note however that by directly following the measurement as a setpoint change, only the integral mode will act on the deviation, which may entail in a degradation in the response[29], but because we are feeding the output signal from this outer controller into the internal controller running in the firmware of the Crazyflie 2.0, the PonM helps in reducing the oscillations around the setpoint and gives a smoother response. The results of using proportional-on-measurement versus calculating the P-term by the error can be seen in figure 5.8 below.



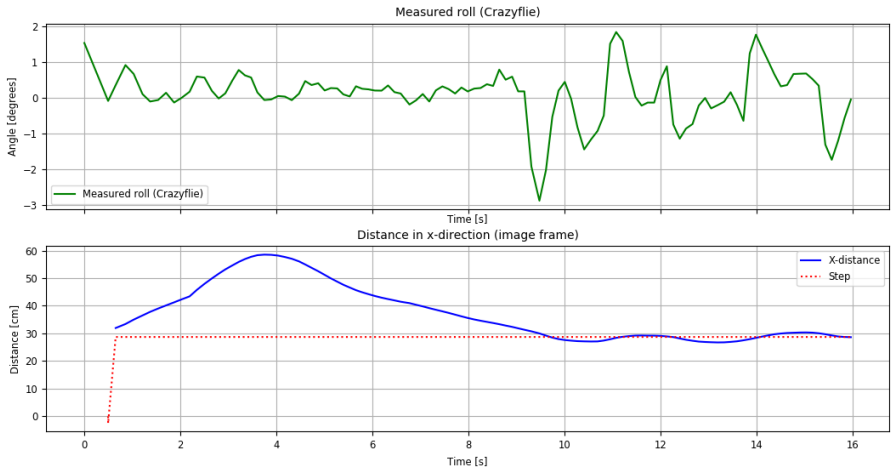
(a) Without PonM



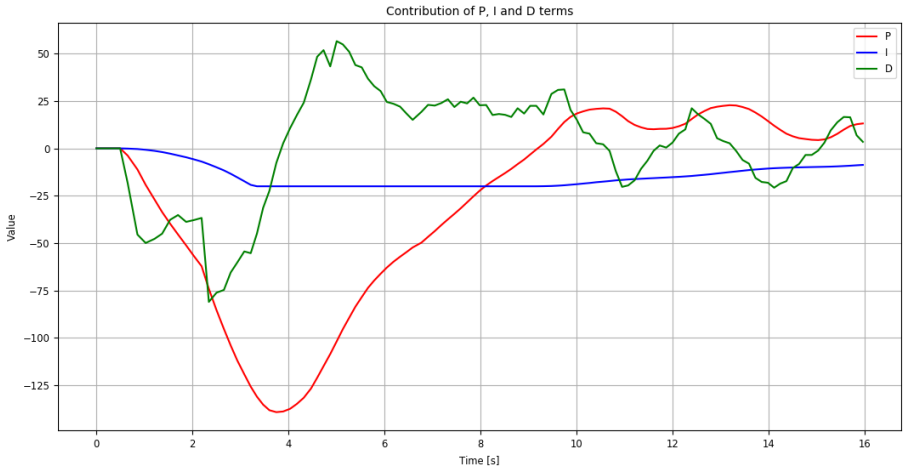
(b) With PonM

Figure 5.8: Step response displaying the results of using proportional-on-measurement versus proportional-on-error for control in x-direction. The gains were defined as: $K_p = 5.10$, $K_i = 5.40$, $K_d = 0.62$ for both experiments.

Finally, by using the two PID controllers for the yaw-angle and translation in x-direction in parallel to each, an experiment was conducted where the quadcopter was placed in a position approximately 30 centimeters away from the marker in x-direction with a 35° relative angle ($\Psi_{cmd} = 35°$). By having the two controller work together in order for the MAV to orient and position itself such that the center point of the marker lies in the center of the camera reference frame in x-direction and the y-axis of the MAV lies collinear with the z-axis of the marker reference frame, the results exhibit quite stable operation and the quadcopter is able to reach the target setpoints within a reasonable time.



(a) Step response for control in x-direction



(b) P,I,D term contributions to x-direction controller

Figure 5.9: Results of x-direction controller when working together with the controller for the yaw-angle.

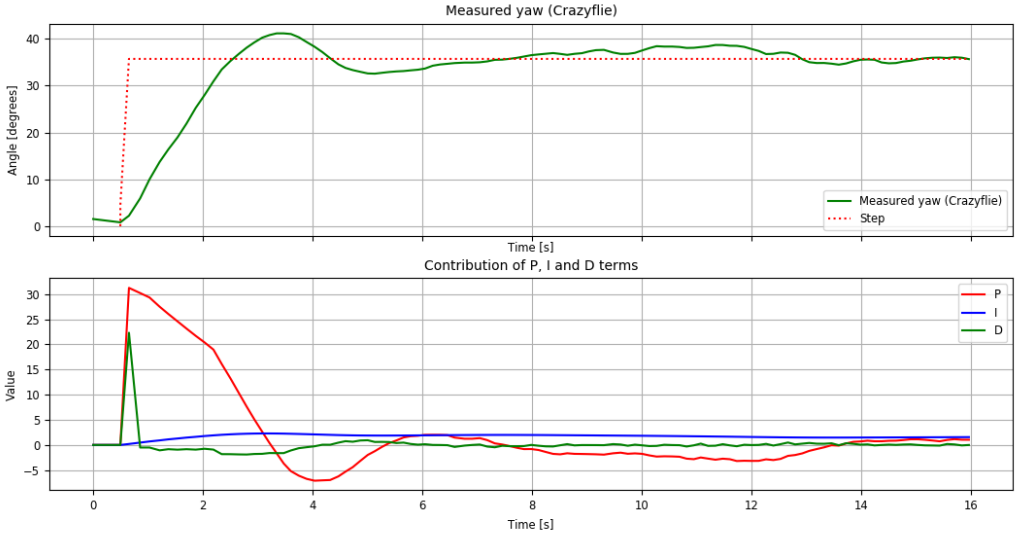


Figure 5.10: Results of yaw-angle controller when working together with the controller for motion in x-direction

The results when the two controllers are working together can be considered to be quite accurate with a deviation around the setpoint Ψ_{SP} of only about ± 5 degrees. The x-direction controller shows significant improvement over using a simple P-controller but still displays a minor oscillation around the setpoint despite the use of PonM. However, the results are very much sufficient for the scope of this project as it shows that the MAV is able to orient and position itself relative to the target marker with only a small amount of oscillations. The relative translation in marker z-direction is controlled by a straightforward P-controller which simply computes the error signal and sends the proportional error term to the internal controller of the MAV. Because of the simple nature of the z-direction controller, the tuning process was quite uncomplicated due to that the MAV does not have to continuously correct itself relative to the marker in z-direction until it has locked on target as described in section 4.9.

One way of further improving the performance of the outer controllers in the system could be to include a feed forward term in the control loop as:

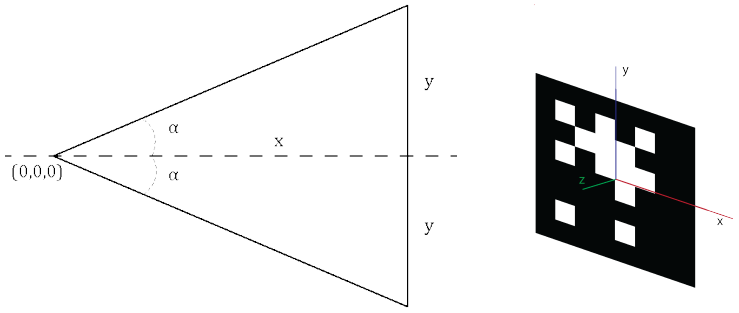
$$u(t) = \ddot{x}_{des} + K_d\dot{e}(t) + K_p e(t) + K_i \int e(t)dt \tag{5.2}$$

, which can aid in reducing the error more quickly, thus improving the speed of the response and leave less of an error over time, though this lies in the scope of future work and will not be evaluated during this project. Also, the performance of the control loop is dependent on the computational capacity of the computer since the detection and pose estimation algorithm requires a significant amount of processing time. By reducing the processing time required in the control loop, the overall performance can be improved since it would facilitate for an increased number of samples per second.

5.4 Detection performance

The field of view (FOV) of the camera was calculated simply by placing the camera perfectly centred and square and facing a wall with two registration grids, placed level in height with respect to the camera and each other. By accurately measuring the distance between the two grids and the distance towards the camera along the principal axis, the field of view can be calculated by simple trigonometry. By knowing the distance, y between the registration grids and the distance, x between the wall and the camera, the angle, α between the camera principal axis and each of the registration grids can be calculated by simply: $\alpha = tan^{-1}\left(\frac{y/2}{x}\right)$, and $FOV = 2\alpha$. The resultant field of view I got from these calculations and measurements was $FOV \approx 120°$, which is slightly less than the 140° specified by the manufacturer of the camera. This could be due to either slightly inaccurate measurements or because the field of view specified by the manufacturer is estimated given other unknown variables.

As an initial experiment for evaluating the performance of the detection algorithm in the ArUco module, an ArUco marker was placed in various angles, $\alpha \in [-60, 60]$ in front of the camera while keeping the distance, $x = 125cm$ as shown in figure 5.11a below constant. This means that we cover the complete field of view of the camera and we can evaluate the ability of the function to output a signal that indicates if the detection and verification of the marker was successful or not. In this case, the detection function did output a signal indicating that the detection was successful for all angles $\alpha \in [-60, 60]$.



(a) Set-up for detecting the marker when varying the angle α and keeping x constant.

(b) ArUco marker with ID=24 from the 6×6 dictionary

Figure 5.11: Experiment configuration

In the second experiment, the angle of rotation about the marker y-axis (see figure 5.11b) was varied in steps of 10°, starting from 0 to 90, while keeping the camera at a constant distance away from the marker at about $x = 75centimetres$, as well as the angle $\alpha = 0°$. The detection and pose estimation algorithm was able to, with good results, estimate the angle of rotation about the marker y-axis with only minor deviations of about ±2° when the detection algorithm was running in real-time and given visual input data as a continuous data stream. The results of the experiment can be found in table 5.3 below.

Table 5.3: Result of marker detection by varying the angle Θ_M about the marker y-axis

Angle Θ_M	Detected	Calculated angle
0°	Yes	0.11°
10°	Yes	10.2°
20°	Yes	21.1°
30°	Yes	31.4°
40°	Yes	40.6°
50°	Yes	51.2°
60°	Yes	61.9°
70°	Barely	70.3°
80°	No	-
90°	No	-

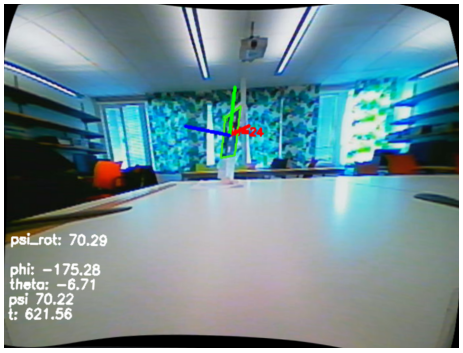


Figure 5.12: Marker detected at an angle of 70°. Beyond this angle, the marker cannot be detected given the camera used in this project. In the figure, psi_rot denotes the angle of the marker, corrected by the relative rotation of the camera.

In this experiment, the camera was placed as centred and square as possible towards the marker, but note that minor deviations in the measurements may have been present due to how the experiment was conducted. The results prove however that the camera is able to detect the marker at aggressive angles up to a maximum of about $\Theta_M = 70°$, at which the detection is lost for numerous amount of frames when the detection algorithm is running in real-time. At low angles, $\Theta_M \in [0, 50]$, the detection of the marker is quite consistent with only a very few detection losses between frames. The calculated angle in table 5.3 is computed through taking the average between the estimated angle in $n = 10$ frames given a successful detection. The experiment was conducted only for angles $0 < \Theta_M < 90$ because negative angles would yield the same result as positive angles and any angle beyond 90 degrees would imply that the identifier of the marker cannot be determined by the computer because the inner binary matrix of the marker becomes non-existent in the image frame.

One should also note that external conditions play a huge role in the performance of the detection algorithm. When the image taken by the camera is being processed in the detection algorithm, it relies on segmentation through thresholding and extraction of contours, meaning that the presence of e.g., specular reflections coming from surrounding light sources or noise and distortions in the image originating from electromagnetic interference in the transmission can disrupt the results. There are many ways of minimizing the influence of these interfering factors. One possible way is to pre-process the image by identifying the disrupting segment in the image and then filter out the noise before it is given as input to the marker detection function. Yet, this may require an additional computational cost which could impact the overall performance of the control loop by increasing the processing time. Therefore, it was decided to instead minimize the influence of external factors by simply conducting the experiments in a controlled environment.

Figure 5.13: Reflections coming from external light sources could impact the ability of the detection algorithm to identify the marker.

Figure 5.13 shows how lighting could interfere with the detection of the marker by creating reflections on the target which obscure its internal binary matrix. The ArUco module provides the possibility of applying error detection and correction techniques through modifying the function parameters that govern the marker identification by bit extraction, but one should bare in mind that increasing the number of bits that can be corrected by the detection function could entail in an increased amount of false positives.

The next experiment for evaluating the performance of the detection algorithm was to examine at which distance, x (as shown in figure 5.11) the marker can be detected when placed directly in front of the camera with zero relative rotation. The experiment was conducted by placing the marker at various distances, $x \in [10, 220]$ metres away from the camera and study the distance computed by the pose estimation algorithm and asses its accuracy and precision, as well as determine the distance interval of which the detection algorithm is able detect and validate the marker. As mentioned in section 4.5, the capability of marker detection and verification algorithm is highly dependent on the size of the internal binary matrix of the marker. A marker with a low amount of bits should theoretically be less prone to erroneous detection and allows for detection at greater distances because of the larger size of the bits in the image frame. In order to keep the results of the experiments consistent, the marker used is characterized by a $200mm \times 200mm$ square with a 6×6 internal matrix and has the identifier $ID = 24$.

All measurements were done in conditions where there was no rotation of the camera with respect to the marker, i.e., ${}^{\mathcal{C}}\mathbf{R}_{\mathcal{M}} = \boldsymbol{I}_{3\times3}$ and the translation vector, ${}^{\mathcal{C}}\mathbf{t}_{\mathcal{M}} = [t_x, t_y, t_z]$ was computed by the ArUco pose estimation function. In this case, we are only interested in the component t_z, which holds the value of the translation along the marker z-axis between the reference frame of the marker and the camera.

Table 5.4: Actual distance between the camera and the marker, versus the distance calculated by the marker detection and pose estimation function

Actual distance [cm]	Detected	Calculated distance [cm]
10	No	-
20	Yes	20.47
25	Yes	25.20
30	Yes	30.07
35	Yes	35.63
40	Yes	40.31
45	Yes	46.09
50	Yes	51.01
55	Yes	56.83
60	Yes	61.32
65	Yes	66.22
70	Yes	71.68
75	Yes	76.58
80	Yes	81.39
85	Yes	87.65
90	Yes	92.81
95	Yes	93.50
100	Yes	109.74
105	Yes	117.08
110	Yes	114.37
115	Yes	118.60
120	Yes	127.14
125	Yes	131.91
130	Yes	133.61
135	Yes	142.08
140	Yes	146.03
145	Yes	152.53
150	Yes	157.52
155	Yes	162.04
160	Yes	168.26
165	Yes	174.42
170	Yes	181.72
175	Yes	185.17
180	Yes	188.28
185	Yes	195.72
190	Yes	204.65
195	Yes	209.36
200	Yes	213.89
205	Yes	214.09
210	Yes	219.63
215	No	-
220	No	-

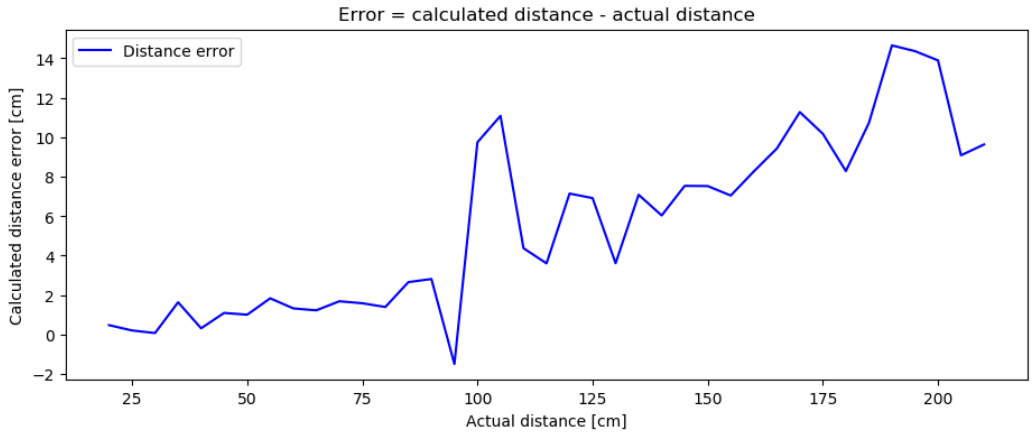


Figure 5.14: The difference between the calclated distance t_z and the actual measured distance

As we can see from the graph in figure 5.14 which holds the values given in table 5.4, at short distances, the pose estimation algorithm is able to estimate the distance between the camera and the marker with only a very small amount of deviation from the actual value (approximately $\pm 2 centimetres$) and the error in the estimated distance increases with the distance. When the marker is placed at the distance limit of detection (about 2 metres), the pose estimation function outputs a value of t_z with a maximum deviation of about 14 centimetres. However, looking at the results from this experiments in terms of absolute units does not tell the whole truth since we have to account for the perspective geometry. A more accurate interpretation of the performance of the estimation in translation would be to consider the error in terms of the ratio between the actual distance and the distance calculated by the pose estimation function: $Error_{\%} = ||100 \cdot (\frac{d_{calculated}}{d_{actual}} - 1)||$

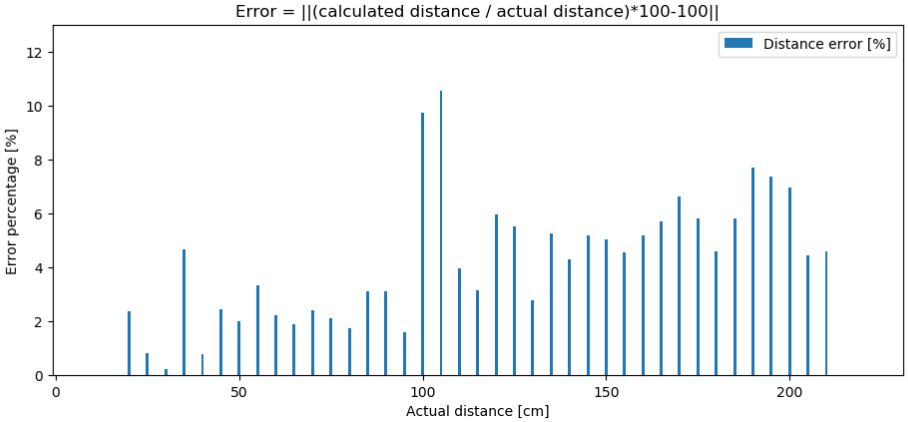
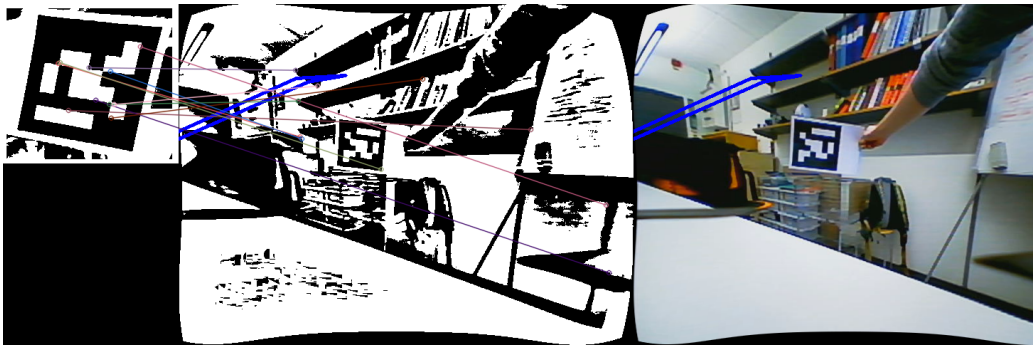


Figure 5.15: The difference between the actual and calculated distance between the camera and the marker in percent

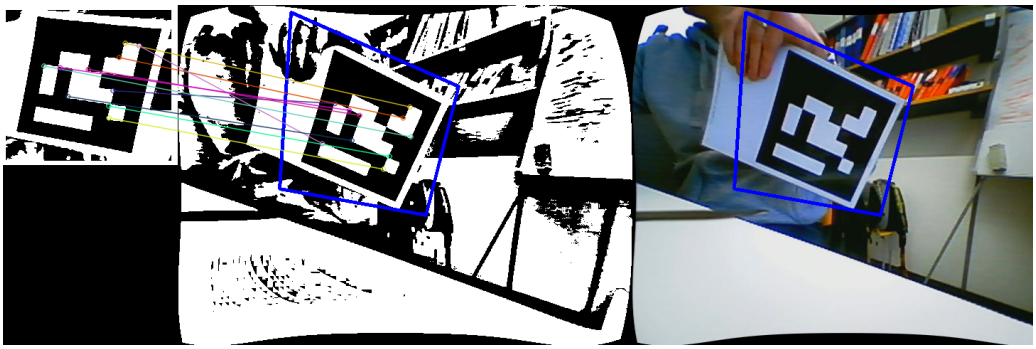
By analysing the results shown in figure 5.15, it can be observed that the error as a percentage follows an upwards trend when the distance between the camera and the marker is increased, similar to when the error was evaluated in terms of absolute units. At the distance of about 1 meter, the error peaks up to around 10%, which is is most probably a computational error that comes as a results of occlusions of the marker in the image frame, making the filter average the current pose of the marker through a broad ranging data set, resulting in a large variance and thus, a larger error.

5.4.1 ORB

As previously mentioned in section 3.4.1, the feature-based detection method ORB was considered an option for detecting the markers in the image frame, as opposed to using the ArUco module for detection and pose estimation. The results from experiments show however that using ORB may not be a feasible option in the case of **this** project because the performance of the detection algorithm when used for detecting codified fiducial markers at various distances and orientations in real time was found to be much lower than for the filter based detection method that comes with the ArUco module. ORB relies on finding and matching features between image frames, hence it requires at least one reference image from which it can relate and link the detected features with the images taken by the camera. Figure 5.16 shows how features can be detected and matched between two images, given both a RGB image (allthough OpenCV uses the BGR convention for historic reasons) and a thresholded binary image.



(a) Failed attempt at detecting the marker through ORB



(b) Successful attempt at detecting the marker through ORB

Figure 5.16: Using a feature-based method for detecting a fiducial marker requires careful adjustment of the detection parameters. The number of detected and matched features directly influence the performance of the vision system.

The coloured lines radiating out from the reference image (aligned to the left in figure 5.16) indicate the matched feature points and the blue rectangle around the marker in the same figure indicate a successful detection. The marker was placed in the vicinity of the camera and the ORB detection function was running in real-time. Immediately, it could be concluded that the ORB detection would not be a viable option in the case of this project as this detection algorithm would require an immense amount of adjustments in order to produce consistent and reliable results. Therefore, it was decided to not put any more effort into improving the detection through ORB, but instead focus on using detection and pose estimation through the ArUco library which had proved to be more reliable and robust than the ORB algorithm.

5.5 Battery life characterization

The Crazyflie 2.0 has a flight time of around 7 minutes as specified on the manufacturer website [2] when it is used in its base configuration, that is without any added peripherals or sensors other than those that comes embedded onto the platform itself. During this project, we augment the capabilities of the MAV to perform relative pose estimation and navigation by adding two additional sensors two the platform, namely the optical flow deck V2 and the camera and transmitter module, which both contribute to the total weight and power consumption of the vehicle. Therefore, it is of interest to study how the addition of these sensors impact the total flight time of the vehicle in order to establish at what battery level the MAV should approach the target and begin the landing procedure.

As an initial experiment, the flight time was measured first without the camera and video transmission module mounted on the vehicle and secondly with the module mounted and turned on, thus contributing to the effective flight time both by its net weight and by the power required for operation. The battery was fully charged and the Crazyflie 2.0 was set to hover in place at a constant height without any commanded inputs to move or change its attitude other than for stabilization purposes. The vehicle was then left in the hovering state until the battery voltage level was measured to $batteryLevel < 3.0V$ whereby a signal was sent to the MAV to initialise a landing procedure and the logged data was stored from memory into a file and the results could be plotted and analysed. The results from measuring the battery level with and without the camera module attached and (turned on) to the quadcopter can be seen in figure 5.17 below.

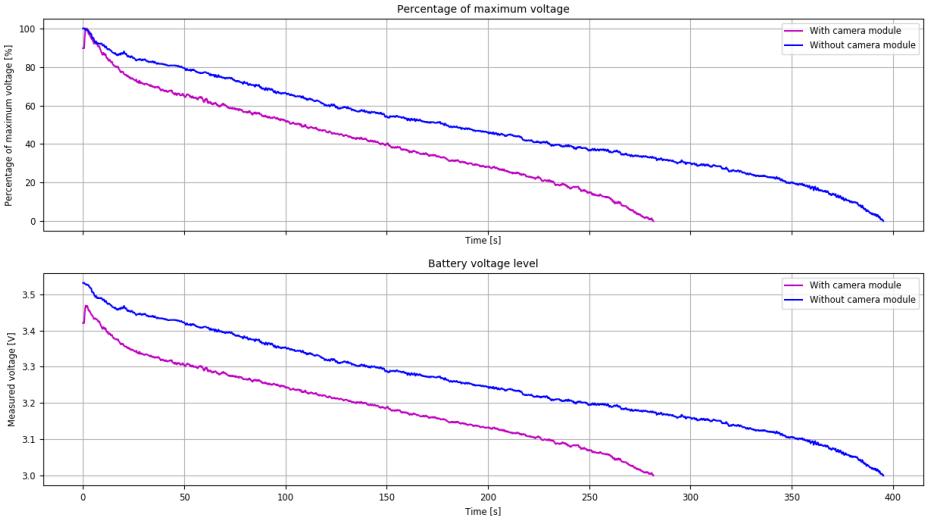


Figure 5.17: Measurements of battery voltage level, with and without the attached camera- and video transmission module. Starting with a fully charged battery and hovering until the voltage between the positive- and negative terminals of the battery has reached 3.0 Volts.

As can be seen upon inspection of the results shown in the graph in figure 5.17, in its base configuration, the Crazyflie 2.0 can hover in place for almost 400 seconds (which equates to roughly 6.7 minutes) until the battery voltage level decreases below 3.0 Volts. It is however possible for the vehicle to remain in flight even for a while longer, but for safety purposes it was decided to let the vehicle land before a critical voltage level is reached where the motors simply cannot produce enough thrust to keep the MAV level and stable.

When the camera and video transmission module is mounted on the Crazyflie 2.0 and turned on, the flight time gets considerably decreased when compared to the flight time in the base configuration. With the module attached and turned on, the flight time is decreased by about 28% through its contribution by power consumption for operation and weight.

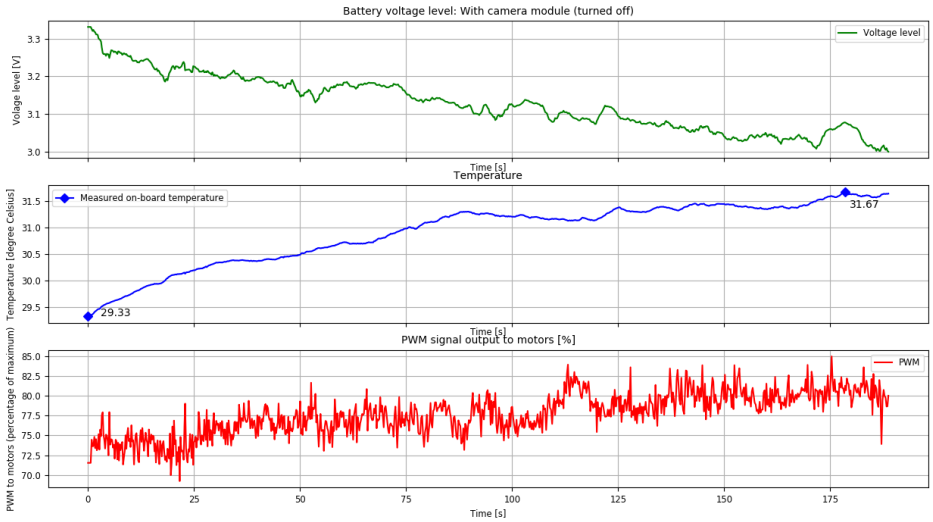


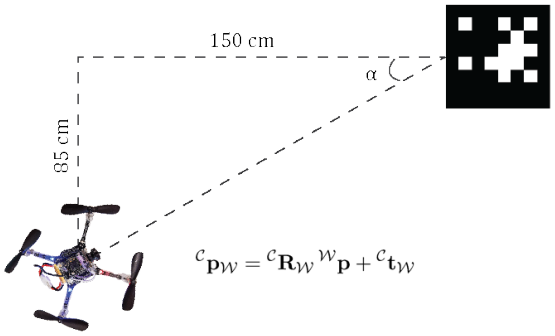
Figure 5.18: Sensor readings during hover with camera and video transmission module on-board (camera and tx-module turned off)

In the figure above (6.1), a similar experiment was conducted but this time with the camera and transmission module mounted on-board the Crazyflie 2.0 and with the power to the module turned off. The intention of this experiment was to assess how the battery life is influenced by only its contribution by weight, excluding the power consumption for operation of the module. However, at this point, the battery had reached too many charge cycles to be able to maintain a similar operational time as for the previous experiments, making the battery discharge more rapidly and thus decreasing the effective flight time substantially. Therefore, we cannot compare the results between the different measurements directly, but we can observe how other parameters influence the flight time.

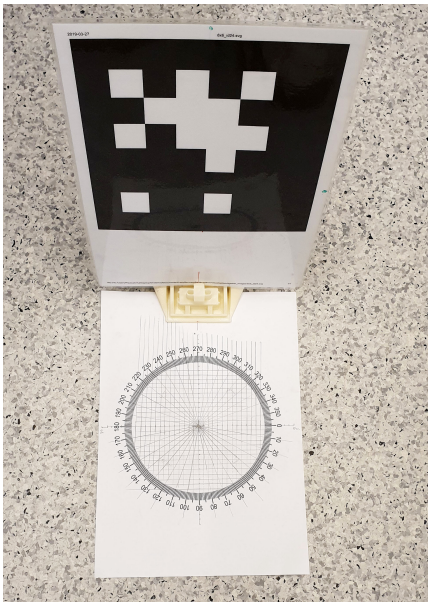
As the battery level decrease, the duty cycle of the PWM signal given as input to the MOSFET-transistors which drive the motors must increase in order to keep the vehicle in equilibrium at the specified height because the thrust generated by each motor must add up to support the total weight of the platform, as described through equation 3.5. The plot in figure 6.1 above shows how the duty cycle of the PWM signal increase as the battery voltage decrease, in order to keep the quadcopter stable and at a constant height. This means that since the thrust output from each motor will determine the position (in this case the height) of the quadcopter, by increasing the cycle frequency, the quadcopter is able to remain in its original pose, but will also entail in an increased power consumption. We can also see from the graph that the readings from the on-board temperature sensor increase with the time of operation. This is interesting because it allows us to reason about the losses in the system that appear as heat losses. The internal resistance in the battery will cause the temperature in the battery to increase over time during operation and causes the voltage to drop, however, since the sensor measures the ambient temperature, we cannot conclude that the increase in temperature is subject to only the heat generated by the battery, but it may also originate from e.g., friction, ohmic or other losses from the brushless motors which generate heat.

5.6 Targeted landing

In order to determine the performance of the vision control system for targeted landing, an experiment was conducted where the marker was placed along a certain distance away from the MAV and with a constant angle, α as shown in figure 5.19a. Both the marker and the quadcopter was placed on the same height and a signal was transmitted to the MAV to take off and hover, whereby it captures visual data from the camera, detects and validates the marker and calculates its current relative pose to the marker. The goal of this experiment was to evaluate the accuracy and precision of the control system by utilizing the visual control input for autonomous navigation towards the detected target and landing in the vicinity of the marker.



(a) The Crazyflie 2.0 was placed at a distance of 1.5 metres away from the marker, pointing directly to the center of the marker with an angle $\alpha = 30^{\circ}$ between the y-axis of the camera reference frame and the z-axis of the marker reference frame.



(b) ArUco marker with ID=24 from the 6×6 dictionary used as target. Below the marker lies a measurement platform on which the MAV should land as close to the center as possible.

Figure 5.19: Experiment configuration

Given the results from the previous experiments, we have concluded that the vision system is able to detect and validate the marker in the image frame, as well as compute the relative pose of the MAV, even with aggressive marker angles assuming favourable lighting conditions. The limitations of the vision system is directly dependent on the capabilities of the camera, so the experiment has to be confined to the capacity of the camera. Therefore, the perpendicular distance between the MAV and the marker was set to $x = 150$ centimetres and the relative angle of rotation about the marker y-axis, ${}^{\mathcal{C}}\boldsymbol{R}(\Psi)_{\mathcal{M}} = \alpha = 30^{\circ}$, as shown in figure 5.19a.

When the quadcopter has found and validated the marker in the image frame, it aligns itself such that its y-axis lies collinear with the z-axis of the marker (thus controlling its yaw-angle, Ψ) and continues with controlling its position in x-direction by moving so that the marker center point always lies in the center of the image frame. If the MAV is able to stay locked on target for one second, it calculates its relative translation in marker z-direction, t_z and begins to approach the marker until it has reached 15 centimetres away, whereby it begins to descend and finally land on the platform beneath the marker as shown in figure 5.19b. By measuring the point of where the MAV landed in relation to the desired landing location (indicated by

the center point of the circular angle diagram), we can through these measurements argue about the accuracy of the vision system and its performance.

The experiment was conducted with five iterations, keeping the distance x and the angle, α as close to constant as possible between each reiteration. The position as well as the attitude of the MAV post landing was carefully measured and logged. The results of each observation can be seen in table 5.5 below, where the column $x > 0$ indicates the offset in x-direction from the desired landing location in positive planar x-direction, $y > 0$ indicate the offset in positive planar y-direction and Ψ specifies the relative angle of rotation about the z-axis of the body fixed frame of the quadcopter.

Table 5.5: Results of targeted landing experiment. The mean value μ and the standard deviation, σ is shown below the table for each respective measurement.

Distance from platform origin		
x [cm]	y [cm]	Ψ [°]
-7.7	1.0	0.12
7.3	-7.1	-10.67
3.5	-3.4	5.19
-6.1	0.5	0.38
3.6	9.2	1.44
$\mu = 0.12$	$\mu = 0.04$	$\mu = -0.708$
$\sigma = 5.644$	$\sigma = 5.438$	$\sigma = 5.301$

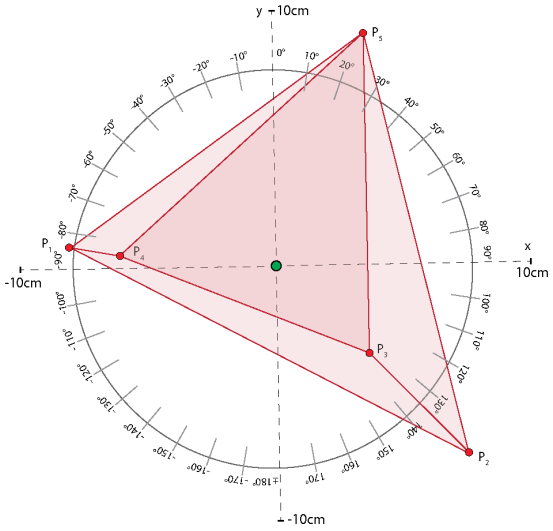
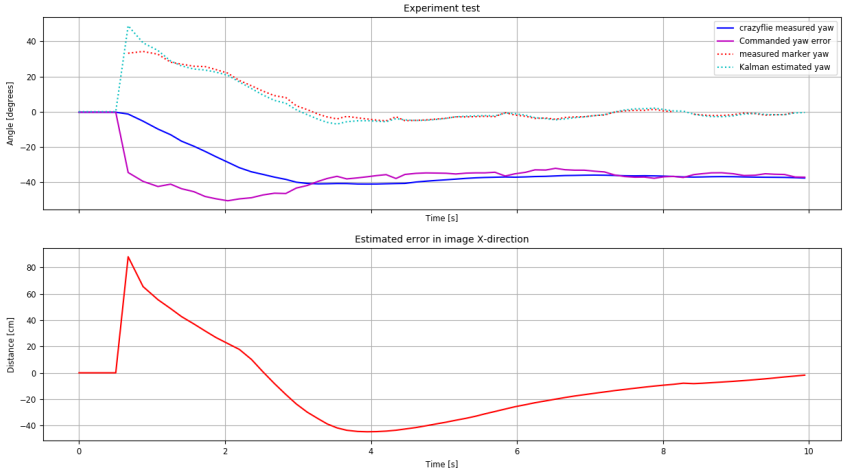


Figure 5.20: Target landing diagram. Each point, P_i on the diagram correspond to the landing point of the MAV for each attempt.

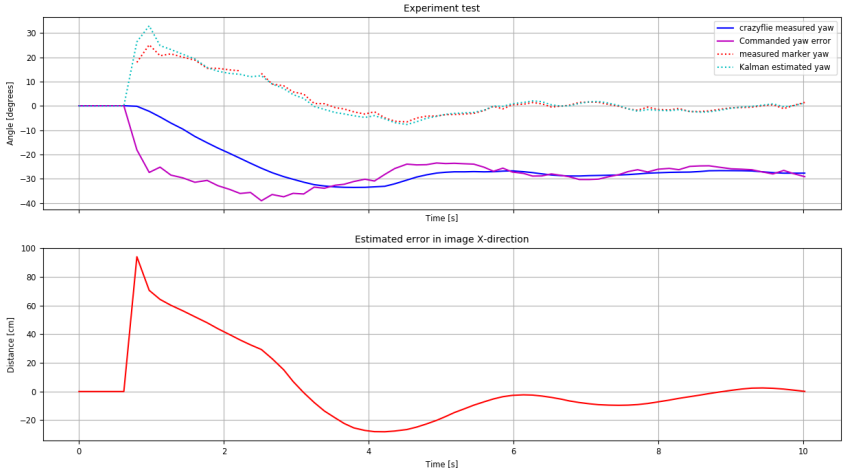
From the results of these five iterations, we can conclude that there are some inconsistencies in terms of both precision and accuracy. The MAV is able to land in relatively close proximity to the center of the landing platform with a standard deviation of about $\sigma \approx 5.5$ centimetres in both x- and y-direction as indicated in figure 5.20, but the worst result resulted in the quadcopter almost crashing into the target with only about five centimeters to spare. A better camera would most probably yield a more accurate and consistent result, but considering that the camera is at best a hobby-grade component, the results from this experiment show that it is possible to achieve autonomous navigation and landing by using cheap and accessible equipment.

For safety reasons and due to the small confined space where the experiment was conducted, the velocity was limited to about 0.08 meters per second for translational movement along all three axes of the MAV and the velocity for the relative attitude of the MAV was limited to a maximum of 90 degrees per second. This means that the time it takes for the vehicle to change its orientation and position relative to the marker is governed by the maximum allowed velocity, the distance it should travel and the required change in attitude. Figure 5.21 below shows how the vehicle changes its position and orientation and the time required for making such a change. The blue line in the plots display the measued yaw-angle of the MAV, the line in magenta shows the yaw-angle and the dotted lines indicate the computed and estimated relative yaw-angle. The line red in displays the difference between the computed (and commanded) distance in x-direction which is the actual distance between the marker in the

image frame. As time passes, the vehicle perform corrections in order to make the red line, as well as the dotted lines converge to zero.



(a) Log-data from attempt number 4



(b) Log-data from attempt number 5

Figure 5.21: During the experiment, data containing information about the actual yaw-angle, ${}^{C}\Psi$, as well as the commanded yaw-angle and the translation in z-direction was logged. The plots show each data-set as a function of time when the vehicle positions and orients itself towards the target marker before it begins to approach the target at $t = 10s$.

From these plots, we can see that it takes about 8-10 seconds for the vehicle to correct its position such that the marker lies in the center of the image frame and approximately 3 seconds for the vehicle to orient itself towards the marker. Also, by looking at the commanded yaw-angle and translation in x-direction, we notice a slight overshoot. This is unfortunately not something I have been able to fully eliminate, but can theoretically be corrected by improving the control system by e.g., increasing the sampling frequency of the data given as input to the controller or by further tune the PID controllers as explained in section 5.3. Due to the time limit of this project, improving the vision control system any further will be left to future work.

Chapter 6

Conclusion and future work

6.1 Conclusion

In this thesis, we presented a vision based control system for the Crazyflie 2.0, incorporating a small camera and wireless video transmitter with which we have demonstrated a method for achieving autonomous navigation and landing towards a fiducial marker. The vision system was designed and implemented with a sub-goal of detecting and validate a target marker in the image frame, for which purpose the ArUco module was used. From the images containing the detected marker, the relative pose of the MAV could be computed in real-time through a homography transformation, which allows the vehicle to align itself towards the marker and finally approach it and land.

The vision control system was implemented off-board, meaning that all computations related to the vision system are done on a ground station computer which receives the video feed from the on-board camera and handles all the heavy computations. The computer then sends the control signals to the MAV, containing information about its relative position and orientation towards the target marker, whereby the input control signal is handled by the on-board controllers. In order to improve the stabilization of the vehicle whilst in flight, the Optical Flow Deck V2 was used. This deck uses a downwards-pointing image sensor which track the planar movement of the MAV above the ground in order to minimize the drift along its x- and y-axes. The deck also utilize a time-of-flight sensor which help the Crazyflie 2.0 maintain its position along its z-axis by measuring the distance to the ground.

In order to evaluate the performance of the whole system, several experiments were conducted which indicate that the vision system as presented in this thesis will work in practice but also leaves room for future improvements. The impact on the battery life due to the addition of the sensors can be considered quite reasonable considering the small form factor of the vehicle. Because of the small size of the quadcopter, the battery must be dimensioned appropriately as well. The flight time of the Crazyflie 2.0 is about 7 minutes in its stock configuration and by adding the optical flow deck and the camera- and video transmitter module, the flight time gets reduced by about 28%. For a larger vehicle with a larger battery carrying the same sensors, the reduction in effective flight time due to the addition of the sensors would be substantially less. However, the camera was chosen in accordance with the price, size and weight of the chosen platform, hence, it was necessary to choose a camera that was as cheap, light-weight and small as possible. This meant that we must make a compromise in terms of performance of the vision system because a better camera would most certainly be able to improve the overall performance of the whole system due to higher image quality, but could also increase the payload or would be more expensive.

Through the results of the conducted experiments, we can conclude that the implemented vision system allows for the MAV to detect, navigate towards and land in relatively close

proximity of the target marker during indoors conditions. In the experiment discussed in section 5.6, the vehicle was able to locate and land on the desired location within a radius of about 10 centimetres from the center-point of the landing pad given the five attempts. The results show that there is definitely room for further development and improvements, but because of the small size of the vehicle and the relatively cheap components used, the results can be considered quite good and has great potential for usage in various applications, such as remote observation of otherwise inaccessible areas, transportation or search tasks. Therefore, by publishing the source code of the project, we aim to inspire and encourage future development and research around autonomous MAVs. Parts of the main software for the vision system can be found in appendix A and the full source code of the project is released as open source on[17].

6.2 Future Work

During development of the vision based control system for the Crazyflie 2.0, a number of interesting approaches came to mind which may be viable for improving the performance or increasing the functionality of the overall system, which will be further discussed in this section.

In the current state of the project, the system has a few limitations. Firstly, as mentioned in section 5.3, the outer controller makes the quadcopter slightly oscillate around the set-point, even in its steady-state. One way of minimizing or eliminating these oscillations would be to include a feed-forward term in the control output that can be used to estimate the output from the controller without having to wait for the response of the controller, thus reducing the error faster. Another way of improving the controllers would be to increase the performance of the ArUco marker detection algorithm. Since the time required for the detection of the marker and relative pose estimation of the camera directly influence the performance of the controller by lowering the number of measurements that can be made per second, decreasing the computational time would have a direct positive impact on the overall performance of the system. Ideally, all computations should be made on-board, meaning that we can further reduce or eliminate the time required for transmission, reception and format conversion, thus we can also dismiss the need of having a ground station computer, making the system fully autonomous. This would however require significant changes to be made in the case of this platform both in terms of hardware and software because of the limited computational power of the MAV and the application specific libraries and devices used during this project.

The ArUco library is natively written in C++ which is generally considered faster than Python, meaning that using C++ could potentially increase the operational speed of the control system and thereby making the system operate in a more stable and robust manner. In its standard form, the ArUco detection module uses adaptive thresholding to detect the marker in the image frame, but has support for global thresholding as well, meaning that instead of changing the threshold dynamically over the image, it operates with the same threshold across the whole image and thereby significantly decreasing the required computational time. This option is only natively available for the C++ programming language and could therefore not be evaluated during this project due to time restrictions.

Furthermore, it would be very interesting to see how the vision based control system could be extended to larger vehicles. A larger vehicle would be able to carry substantially heavier payloads, making it capable of using superior equipment. As previously mentioned, the low performance of the camera makes the vision system only able to detect the 6×6 ArUco marker within distances of about two metres, but a camera with the capability to produce images of higher quality would yield a higher performance in terms of accuracy, precision and

repeatability, thus extending the possibilities for the system to be used in other application to a greater extent. Another future work line could be to further increase the functionality of the system by e.g., having it take off again after it has landed and recharged its batteries. Bitcraze AB provides a wireless Qi-charging deck that can be directly plugged onto the pin-headers on the Crazyflie 2.0. Through using this expansion board, the vehicle can charge its battery without having to directly connect the battery with a cable, so incorporating this functionality to the system could potentially mean that the vehicle can remain in flight until the battery level has decreased below a certain threshold and then navigate towards the charging pad through the vision based control system, recharge and then take off again.

One could also consider scenarios where the vision system can be used to detect obstacles in the environment around the vehicle. Some key components for achieving fully autonomous flight are that the vehicle must be able to estimate its own position and velocity in the room, be able to compute its control demands based on its current position and target position, map its environment and be able to compute a safe path to go from one point to another. Through using *simultaneous localization and mapping*(SLAM), the MAV can localize itself in the room and create a map of its environment. The ArUco library provides a solution to this problem by a real-time application that can build a map of the environment around the camera through detection of the planar markers[26], and incorporating it into this project would potentially allow the MAV to fully localise itself in the room and thereby be able to compute a path in the world frame.

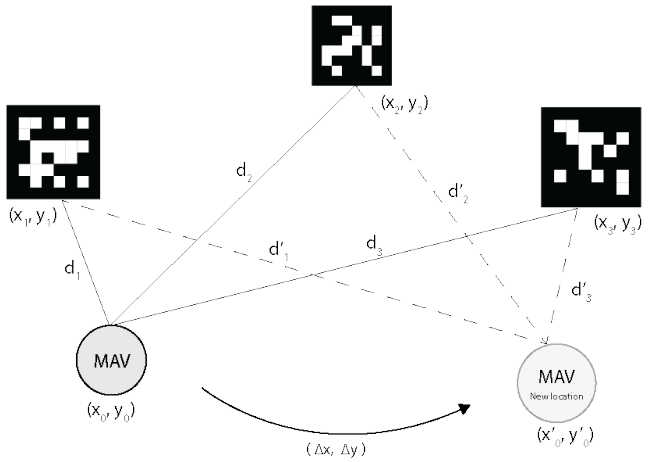


Figure 6.1: SLAM: Simultaneous localization and mapping concept. The general idea is that that the system should be able to localise itself in the room through mapping the location of the markers as well as the velocity of the MAV relative to the markers.

This approach differs from the method presented in this thesis by having the vision system create a map of its environment by which the MAV can determine its position through triangulation , rather than solely rely on position data given by a single detected marker. Due to time restrictions, this method could not be implemented and tested, but is left for future work.

Bibliography

[1] Bitcraze AB. *About Us*. 2019. URL: https://store.bitcraze.io/pages/about-us (visited on 04/04/2019).

[2] Bitcraze AB. *Crazyflie 2.0*. 2019. URL: https://store.bitcraze.io/products/crazyflie-2-0 (visited on 04/04/2019).

[3] Bitcraze AB. *Crazyradio PA 2.4 GHz USB dongle*. 2019. URL: https://store.bitcraze.io/collections/kits/products/crazyradio-pa (visited on 04/04/2019).

[4] Bitcraze AB. *Flow deck v2*. 2019. URL: https://wiki.bitcraze.io/projects:crazyflie2:expansionboards:flow-v2 (visited on 04/04/2019).

[5] Stuart Bennett. "A Brief History of Automatic Control". In: *IEEE Control Systems* 16.3 (1996), pp. 17–25. ISSN: 1066033X. DOI: 10.1109/37.506394.

[6] Oliver Dunkley et al. "Visual-inertial navigation for a camera-equipped 25g nano-quadrotor". In: *IROS2014 aerial open source robotics workshop*. 2014, p. 2.

[7] Eachine. *Eachine M80S M80 Micro FPV Racer RC Drone Spare Parts 5.8G 25MW 48CH VTX 600TVL CMOS 1/3 FPV Camera*. 2019. URL: https://www.eachine.com/Eachine-M80S-M80-Micro-FPV-Racer-RC-Drone-Spare-Parts-5_8G-25MW-48CH-VTX-600TVL-CMOS-1-or-3-FPV-Camera-p-1174.html (visited on 04/04/2019).

[8] Eachine. *Eachine ROTG01 UVC OTG 5.8G 150CH Full Channel FPV Receiver For Android Mobile Phone Smartphone*. 2019. URL: https://www.eachine.com/Eachine-ROTG01-UVC-OTG-5.8G-150CH-Full-Channel-FPV-Receiver-For-Android-Mobile-Phone-Smartphone-p-843.html (visited on 04/04/2019).

[9] Kurt Konolige Ethan Rublee Vincent Rabaud and Gary Bradski. "ORB: An efficient alternative to SIFT or SURF". In: *2011 International Conference on Computer Vision*. 2011, pp. 2564–2571. DOI: 10.1109/ICCV.2011.6126544.

[10] Davide Falanga et al. "Vision-based autonomous quadrotor landing on a moving platform". In: *SSRR 2017 - 15th IEEE International Symposium on Safety, Security and Rescue Robotics, Conference* (2017), pp. 200–207. DOI: 10.1109/SSRR.2017.8088164.

[11] C. Forster, M. Pizzoli, and D. Scaramuzza. "SVO: Fast semi-direct monocular visual odometry". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. 2014, pp. 15–22. DOI: 10.1109/ICRA.2014.6906584.

[12] Julian Forster. "System Identification of the Crazyflie 2.0 Nano Quadrocopter". In: *Thesis* (Aug. 2015). URL: https://polybox.ethz.ch/index.php/s/20dde63ee00ffe7085964393a55a91c7.

[13] Sergio Garrido-Jurado et al. "Generation of fiducial marker dictionaries using Mixed Integer Linear Programming". In: *Pattern Recognition* 51 (Oct. 2015). DOI: 10.1016/j.patcog.2015.09.023.

[14] T. Hägglund and K. J. Åström. "Revisiting The Ziegler-Nichols Tuning Rules For Pi Control". In: *Asian Journal of Control* 4.4 (2002), pp. 364–380. DOI: 10.1111/j.1934-6093.2002.tb00076.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1934-6093.2002.tb00076.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1934-6093.2002.tb00076.x.

[15] Reza N. Jazar. *Theory of Applied Robotics - Kinematics, Dynamics, and Control (2nd Edition)*. Springer, Apr. 2010. ISBN: 9781441917492.

[16] Simon Johansson. "Control of drone with applied weights". Bachelor's Thesis. Karlstad University, June 2018.

[17] Christoffer Karlsson. *Crazyflie-vision-system*. https://github.com/lecristoph/Crazyflie-vision-system. 2019.

[18] Kiam Heong Ang, G. Chong, and Yun Li. "PID control system analysis, design, and technology". In: *IEEE Transactions on Control Systems Technology* 13.4 (2005), pp. 559–576. ISSN: 1063-6536. DOI: 10.1109/TCST.2005.847331.

[19] T. Kurita, N. Otsu, and N. Abdelmalek. "Maximum likelihood thresholding based on population mixture models". In: *Pattern Recognition* 25.10 (1992), pp. 1231 –1240. ISSN: 0031-3203. DOI: https://doi.org/10.1016/0031-3203(92)90024-D. URL: http://www.sciencedirect.com/science/article/pii/003132039290024D.

[20] Dilek Kurtulus. "Introduction to micro air vehicles: concepts, design and applications". In: Apr. 2011, pp. 219–255. ISBN: 978-2-87516-017-1.

[21] Roger Labbe. *Kalman and Bayesian Filters in Python*. https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python. 2014.

[22] OpenCV Labs. *Calibration with ArUco and ChArUco*. 2019. URL: https://docs.opencv.org/3.4/da/d13/tutorial_aruco_calibration.html (visited on 05/07/2019).

[23] Sven Lange, N Sunderhauf, and Peter Protzel. "A vision based onboard approach for landing and position control of an autonomous multirotor UAV in GPS-denied environments". In: *Proceedings of the International Conference on Advanced Robotics* (2009), pp. 1–6. DOI: 10.1109/ROBOT.2001.932859. URL: http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=5174709{\%}5Cnhttp://www.dis.uniroma1.it/lmarchetti/private/quadrotor/lange-vision-based-onboard-approach-landing-position-control-UAV-gps-denied-environments.pdf.

[24] Lili Ma, YangQuan Chen, and Kevin L. Moore. "Rational Radial Distortion Models with Analytical Undistortion Formulae". In: (2003). arXiv: 0307047 [cs]. URL: http://arxiv.org/abs/cs/0307047.

[25] Rafael Munoz-Salinas. *ArUco: One library to rule them all*. 2019. URL: https://bitbucket.connectia.com/users/vfilatov/repos/aruco/browse?at=55d86f631b35b7a80bc1094fb78c98e0d165 (visited on 05/07/2019).

[26] Rafael Muñoz-Salinas, Manuel Marín-Jiménez, and Rafael Medina-Carnicer. "SPM-SLAM: Simultaneous Localization and Mapping with Squared Planar Markers". In: *Pattern Recognition* 86 (Sept. 2018). DOI: 10.1016/j.patcog.2018.09.003.

[27] Brent Perreault. "Introduction to the Kalman Filter and its Derivation". In: (2012), pp. 1–10.

[28] Axel Reizenstein. "Position and Trajectory Control of a Quadcopter Using PID and LQ Controllers". In: (2017).

[29] R.R. RHINEHART. "Control Modes—PID Variations". In: *Process Control* (2013), pp. 30–33. DOI: 10.1016/b978-0-7506-2255-4.50010-6.

[30] J. A. Romero Perez and P. Balaguer Herrero. "Extending the AMIGO PID tuning method to MIMO systems". In: *IFAC Proceedings Volumes (IFAC-PapersOnline)* 2.PART 1 (2012), pp. 211–216. ISSN: 14746670.

[31] Francisco Romero Ramirez, Rafael Muñoz-Salinas, and Rafael Medina-Carnicer. "Speeded Up Detection of Squared Fiducial Markers". In: *Image and Vision Computing* 76 (June 2018). DOI: 10.1016/j.imavis.2018.05.004.

[32] Ethan Rublee, V Rabaud, and K Konolige. "ORB : an efficient alternative to SIFT or SURF About local feature and matching Motivation oFAST – Oriented FAST BRIEF (Calonder et al . 2010)". In: *Intl. Conf. Computer Vision* (2011), pp. 1–5. ISSN: 00043591. DOI: 10.1002/art.38045. arXiv: 0710.1980.

[33] D. Scaramuzza and F. Fraundorfer. "Visual Odometry [Tutorial]". In: *IEEE Robotics Automation Magazine* 18.4 (2011), pp. 80–92. ISSN: 1070-9932. DOI: 10.1109/MRA.2011.943233.

[34] Bruno Siciliano et al. *Robotics: modelling, planning and control*. Springer, 2009. ISBN: 9781846286414.

[35] OpenCV dev team. *Camera calibration With OpenCV*. 2019. URL: https://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html (visited on 04/23/2019).

[36] Bertil Thomas. *Modern reglerteknik*. Liber, 2016. ISBN: 9789147112128.

[37] Michael Veth. "Fusion of Imaging and Inertial Sensors for Navigation". In: (Jan. 2006).

[38] Greg Welch and Gary Bishop. "TR 95-041: An Introduction to the Kalman Filter". In: *In Practice* 7.1 (2006), pp. 1–16. ISSN: 10069313. DOI: 10.1.1.117.6808. arXiv: arXiv:1011.1669v3. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.79.6578{\&}rep=rep1{\&}type=pdf.

[39] Stefano Soatto Yi Ma Jana Kosecka and Shankar Sastry. "An Invitation to 3-D Vision - From Images to Models". In: (Nov. 2001).

Appendices

Appendix A

Main application (Python)

```
import logging
import time
import numpy as np
import cv2
import yaml
import math
import sys
import os
import traceback
from simple_pid import PID
from filterpy.kalman import KalmanFilter

import cflib.crtp
from cflib.crazyflie import Crazyflie
from cflib.crazyflie.syncCrazyflie import SyncCrazyflie
from cflib.positioning.motion_commander import MotionCommander
from cflib.crazyflie.log import LogConfig
from cflib.crazyflie.syncLogger import SyncLogger

import arucoCam as arc
from plotData import plotData

err_dict = {
    "1": "Could not reshape. Multiple markers detected ?",
    "2": "No Crazyflies found, cannot run test",
    "3": "Something went wrong. Closing webcam.",
    "4": "Battery level is low. landing...",
    "5": "Unknown error. Check traceback:"
}


def getData():
    with SyncLogger(scf, lg_stab) as logger:
        for log_entry in logger:
            data = log_entry[1]
            yaw = data.get('stabilizer.yaw')
            pitch = data.get('stabilizer.pitch')
            roll = data.get('stabilizer.roll')
```

```
            vbat = data.get('pm.vbat')
            time.sleep(0.02)
            return(yaw, pitch, roll, vbat)


def getCFparams(scf, available):
    try:
        yaw_CF, pitch_CF, roll_CF, vbat = np.asarray(getData()).tolist()
    except TypeError:
        print("Error: ", sys.exc_info()[0])
        print(TypeError)
        traceback.print_exc()

    if np.size(vbat) != 0:
        return(yaw_CF, pitch_CF, roll_CF, vbat)
    else:
        return(0, 0)


def angCor(alpha):
            if alpha > 0:
                alpha = 180 - alpha
            else:
                alpha = abs(alpha) - 180
            return(alpha)


def getMarkerAngles(cam):
        # Read image frame from camera
        [ret, img] = cam.read()  # Take snapshot from camera
        # Undistort the frame
        img = arc.undistort(img, cMatrix, k_dist)
        # Detect ArUco marker
        # corners = [x1, y1; x2, y2; ..., x4, y4] where x1,y1 = top left corner
        corners, rvec, tvec, ids, imgWithAruco = arc.ArUcoDetect(img, cMatrix,
                                                                 k_dist, m_len,
                                                                 aruco_dict,
                                                                 arucoParams)
        # Get angles between camera and marker with ID=2
        if ids is not None and id2find in ids:
            idx_r, idx_c = np.where(ids == id2find)
            tvec = tvec[idx_r, :]
            rvec = rvec[idx_r, :]
            # draw rectangle around marker with ID=2
            corners = np.asarray(corners)
            corners = corners[idx_r, :]
            corners = np.reshape(corners, [4, 2])
            imgWithAruco = arc.drawRec(imgWithAruco, corners)
            EulerAngles = arc.angleFromAruco(rvec, tvec, ids)
        else:
            EulerAngles = None
```

```
        if EulerAngles is not None:
            psi = EulerAngles[0]*180/math.pi  # yaw
            theta = EulerAngles[1]*180/math.pi  # pitch
            phi = EulerAngles[2]*180/math.pi  # roll
            # Corrent for rotation
            psi = angCor(psi)
            EulerAngles = np.array([psi, theta, phi])
            EulerAngles_rot = arc.rotate(EulerAngles)
            alpha = EulerAngles_rot[0, 0]  # yaw
            return(alpha, imgWithAruco, tvec)
        else:
            return(None, imgWithAruco, None)


class CF:
    def __init__(self, scf, available):
        # get yaw-angle and battery level of crazyflie
        self.psi, self.theta, self.phi, self.vbat = getCFparams(scf, available)
        self.scf = scf
        self.available = available
        self.mc = MotionCommander(scf)
        self.psi_limit = 0.7    # Don't cmd an angle less than this [deg]
        self.des_angle = 0  # Set to zero initially

    def update(self, des_angle, eul, turnRate):
        if 'psi' in eul and abs(des_angle) > self.psi_limit:
            self.des_angle = des_angle
            if des_angle > 0:
                if not self.mc.turn_right(abs(self.des_angle), turnRate):
                    pass
            else:
                if not self.mc.turn_left(abs(self.des_angle), turnRate):
                    pass
        # Compute current angle (yaw) of CF
        self.psi, self.theta, self.phi, self.vbat = getCFparams(self.scf,
                                                                 self.available)
        return(self.psi, self.theta, self.phi, self.vbat)

    # Move cf left or right
    def update_x(self, dist):
        if dist is not None and dist != 0:
            if not self.move_distance(0, -dist, 0):
                pass
        elif dist is not None and dist == 0 and self.des_angle == 0:
            self.mc.stop()

    def takeoff(self):
        self.mc.take_off()

    def land(self):
        self.mc.land()
```

```
    def move_distance(self, x, y, z):
        '''
        Move in a straight line, {CF frame}.
        positive X is forward [cm]
        positive Y is left [cm]
        positive Z is up [cm]
        '''
        velocity = 0.08
        x = x/100
        y = y/100
        z = z/100

        distance = math.sqrt(x**2 + y**2 + z**2)

        if x != 0:
            flight_time = distance / velocity
            velocity_x = velocity * x / distance
        else:
            velocity_x = 0
        if y != 0:
            velocity_y = velocity * y / distance
        else:
            velocity_y = 0
        if z != 0:
            velocity_z = velocity * z / distance
        else:
            velocity_z = 0

        self.mc.start_linear_motion(velocity_x, velocity_y, velocity_z)
        if x != 0:
            time.sleep(flight_time)
        return(False)


if __name__ == '__main__':
    # Define radio interface
    URI = 'radio://0/80/2M'
    # Initialize camera
    cam = cv2.VideoCapture(1)
    # Name of plot figure output
    figname = "experimentData1.png"
    # Title of plot
    figtitle = "Experiment_test"
    # Name of dumpfile
    dumpfile_name = "testData1.yaml"
    # Do we want live display or not ?
    dispLiveVid = True
    # Marker side length [cm]
    m_len = 19.0
    # Marker ID to find:
```

```
id2find = 24
# Init write VideoWriter
FILE_OUTPUT = 'output.avi'
# Checks and deletes the output file
# You cant have a existing file or it will through an error
try:
    if os.path.isfile(FILE_OUTPUT):
        os.remove(FILE_OUTPUT)
except PermissionError:
    print("Error: ", sys.exc_info()[0])
    traceback.print_exc()

# Get width of camera capture
width = (int(cam.get(3)))
# Get height of camera capture
height = (int(cam.get(4)))
# Define some text parameters
font = cv2.FONT_HERSHEY_SIMPLEX
s1 = (10, int(height-40))
s2 = (10, int(height-60))
s3 = (10, int(height-80))
s4 = (10, int(height-100))
s5 = (10, int(height-120))
s6 = (10, int(height-140))
fontScale = 0.6
fontColor = (255, 255, 255)  # white
lineType = 2
# Define the codec and create VideoWriter object
fourcc = cv2.VideoWriter_fourcc(*'XVID')
out = cv2.VideoWriter(FILE_OUTPUT, fourcc, 8.0, ((width), (height)))
# Open camera calibration data
with open('calibration.yaml') as f:
    calParams = yaml.load(f)
# Load camera matrix and distortion coeffs
cMatrix = calParams.get('camera_matrix')
k_dist = calParams.get('dist_coeff')
# Convert to numpy arrays
cMatrix = np.array(cMatrix)
k_dist = np.array(k_dist)
# keep track of values for plotting
setpoint = np.array([])
yaw_cf = np.array([])
pitch_cf = np.array([])
roll_cf = np.array([])
ar_time = np.array([])
ar_bat = np.array([])

# Set up aruco dictionary and parameters
aruco_dict, arucoParams = arc.createAruco()

# Init Kalman filter
```

```
x = 8   # n.o. elements in state vector
z = 4   # n.o dimensions in state vector
f = KalmanFilter(dim_x=x, dim_z=z)  # Yaw filter
# Uncertainty factor
unc_f = 0.04
# Define Kalman filter state estimate vector
f.x = np.zeros((x, 1), dtype='float')
# state transition matrix
f.F = (np.eye(x))
f.F[0, 1] = 1.0
f.F[2, 3] = 1.0
f.F[4, 5] = 1.0
f.F[6, 7] = 1.0
# Measurement function
f.H = np.zeros((z, x), dtype='float')
# Define elements to which we perform calculations
f.H[0, 0] = 1.0
f.H[1, 2] = 1.0
f.H[2, 4] = 1.0
f.H[3, 6] = 1.0
# state uncertainty
f.R *= unc_f
# process noise
f.Q *= unc_f*0.1

low_battery = 2.71  # Don't take of if battery level < low_battery [V]
nf_max = 8  # Define maximum number of estimated frames in a row
nf = nf_max + 1  # Define current number of estimated frames
alpha = 0.91  # Inertial factor for velocity degregation
samples_max = 10   # Define sample heap size for moving avg filter
zs = np.array([])   # Array for moving average filter (yaw)
ds = np.array([])   # Array for moving average filter (transl_z)
xs = np.array([])   # --||-- (transl_X = right/left in img)
ys = np.array([])   # --||-- (tranls_Y = height)
arb = np.array([])  # Array for moving avg filter (battery level)
r = np.array([0])  # Array for storing location and velocity estimates(yaw)
r2 = np.array([0])  # --||-- (transl_z)
r3 = np.array([0])  # --||-- (transl_x)
r4 = np.array([0])  # --||-- (transl_y)

# PID Gains
Kp_psi = 1.0  # 1.40
Kd_psi = 0.045  # 0.15
Ki_psi = 0.1  # 0.55
Kp_x = 5.1  # 7.24
Kd_x = 0.62  # 0.46
Ki_x = 5.40  # 0.13.89
turnRate = 360/3.8    # 360/4.5 is default value

# Only output errors from the logging framework
logging.basicConfig(level=logging.ERROR)
```

```
# Initialize the low-level drivers (don't list the debug drivers)
cflib.crtp.init_drivers(enable_debug_driver=False)
# Scan for Crazyflies and use the first one found
print('Scanning interfaces for Crazyflies...')
available = cflib.crtp.scan_interfaces()
print('Crazyflies found:')
for i in available:
    print(i[0])

if len(available) == 0:
    print(err_dict["2"])
else:
    cf = Crazyflie(rw_cache='./cache')
    # Add logging config to logger
    lg_stab = LogConfig(name='Stabilizer', period_in_ms=12)
    lg_stab.add_variable('pm.vbat', 'float')
    lg_stab.add_variable('stabilizer.yaw', 'float')
    lg_stab.add_variable('stabilizer.pitch', 'float')
    lg_stab.add_variable('stabilizer.roll', 'float')
    with SyncCrazyflie(URI) as scf:
        # Define motion commander to crazyflie as mc:
        cf2 = CF(scf, available)
        cf2_psi = cf2.psi    # Get current y,p,r-angles of cf
        cf2_theta = cf2.theta
        cf2_phi = cf2.phi
        cf2_bat = cf2.vbat  # Get current battery level of cf
        marker_psi = 0  # Set marker angle to zero initially

        pid = PID(0, 0, 0, setpoint=marker_psi)  # Initialize PID
        pid_x = PID(0, 0, 0, setpoint=0)  # init PID for roll
        # Define pid parameters
        pid.tunings = (Kp_psi, Kd_psi, Ki_psi)
        pid_x.tunings = (Kp_x, Kd_x, Ki_x)
        pid_x.sample_time = 0.05    # update pid every 50 ms
        pid.output_limits = (-60, 60)
        pid_x.output_limits = (-20, 20)
        pid.proportional_on_measurment = False
        pid_x.proportional_on_measurment = False

        if cf2_bat >= low_battery:
            cf2.takeoff()    # CF takes off from ground
            print("Taking off! Battery level:" + str(cf2_bat))
            time.sleep(1.5)   # Let the CF hover for 1.5s

        t_init = time.time()
        endTime = t_init + 10   # Let the CF do its thing for some time

        # Add values to array for logging/plotting
        setpoint = np.append(setpoint, marker_psi)  # Commanded yaw
        yaw_cf = np.append(yaw_cf, cf2.psi)  # Actual yaw of CF
        pitch_cf = np.append(pitch_cf, cf2.theta)  # Actual pitch of CF
```

```
roll_cf = np.append(roll_cf, cf2.phi) # Actual roll of CF
ar_time = np.append(ar_time, 0)     # Time vector
ar_bat = np.append(ar_bat, cf2_bat)

A = scf.is_link_open()
while time.time() < endTime and cf2_bat >= low_battery and A:
    # Get angle of marker
    try:
        # Get marker yaw-angle, img and translation vector(xyz)
        marker_psi, img, t_vec = getMarkerAngles(cam)

        # Get translational distance in Z-dir, marker->lens
        if t_vec is not None:
            try:
                t_vec = np.asarray(t_vec).tolist()
                t_vec = np.reshape(t_vec, [1, 3])
                dist_x = t_vec[0, 0]
                dist_y = t_vec[0, 1]
                dist_z = t_vec[0, 2]
            except ValueError:
                print(err_dict["1"])
        else:
            dist_z = None
            dist_x = None
            dist_y = None

        # Moving average filter
        if marker_psi is not None:
            zs = np.append(zs, marker_psi)
            ds = np.append(ds, dist_z)
            xs = np.append(xs, dist_x)
            ys = np.append(ys, dist_y)
            marker_psi = np.average(zs)
            dist_z = np.average(ds)
            dist_x = np.average(xs)
            dist_y = np.average(ys)
            if np.size(zs) >= samples_max:
                zs = np.delete(zs, 0)
                ds = np.delete(ds, 0)
                xs = np.delete(xs, 0)
                ys = np.delete(ys, 0)

        # perform kalman filtering
        if marker_psi is not None:
            f.update(np.array([marker_psi,
                               dist_z, dist_x, dist_y]))
            f.predict()
        else:
            f.update(marker_psi)
            f.predict()
```

```
            # Set inertial degregation to velocity for each iteration
            if marker_psi is None and nf < nf_max:
                # f.x holds the estimated values
                f.x[1, 0] *= alpha
                f.x[3, 0] *= alpha
                f.x[5, 0] *= alpha
                f.x[7, 0] *= alpha
                yaw_cmd = f.x[0, 0]  # commanded yaw
                transl_z_cmd = round(f.x[2, 0], 2)  # comm transl_z
                transl_x_cmd = round(f.x[4, 0], 2)  # comm tranls_x
                transl_y_cmd = round(f.x[6, 0], 2)  # comm transl_y
                nf += 1
            elif marker_psi is not None and marker_psi != 0:
                nf = 0
                yaw_cmd = marker_psi  # commanded yaw
                transl_z_cmd = round(dist_z, 2)   # commanded transl_z
                transl_x_cmd = round(dist_x, 2)   # commanded transl_x
                transl_y_cmd = round(dist_y, 2)   # commanded transl_y
            else:
                yaw_cmd = 0
                transl_z_cmd = 0
                transl_x_cmd = 0
                transl_y_cmd = 0

        except (cv2.error, TypeError, AttributeError):
            print(err_dict["3"])
            print("Error: ", sys.exc_info()[0])
            cam.release()
            cv2.destroyAllWindows()
            cf2.land()  # Land the CF2
            print(TypeError)
            traceback.print_exc()
            break

        dt = time.time() - t_init  # compute elapsed time

        # print text on image
        cv2.putText(img, "t: "+str(round(dt, 2)),
                    s1,
                    font,
                    fontScale,
                    fontColor,
                    lineType)
        cv2.putText(img, "psi: "+str(round(yaw_cmd, 2)),
                    s2,
                    font,
                    fontScale,
                    fontColor,
                    lineType)
        cv2.putText(img, "Z: "+str(transl_z_cmd),
                    s3,
```

```
                font,
                fontScale,
                fontColor,
                lineType)
    cv2.putText(img, "X: "+str(transl_x_cmd),
                s4,
                font,
                fontScale,
                fontColor,
                lineType)
    cv2.putText(img, "Battery: "+str(round(cf2_bat, 2)),
                s5,
                font,
                fontScale,
                fontColor,
                lineType)

    if dispLiveVid:
        cv2.imshow('image', img)     # display current frame

        if cv2.waitKey(25) & 0xFF == ord('q'):
            cf2.land()
            cv2.destroyAllWindows()
            break

    out.write(img)  # write img frame to video file
    # Send command signal to CF2 and update cf2_psi
    # 'psi' = yaw
    # 'theta' = pitch
    # 'phi' = roll
    # 'z' = forwards/backwards
    #  ** Send signal to CF + get battery level and yaw-angle
    try:
        if yaw_cmd is not None:
            cval_psi = pid(yaw_cmd)
            cf2_psi, cf2_theta, cf2_phi, cf2_bat = cf2.update(-cval_
                                                              'psi',
        if transl_x_cmd is not None:
            cval_x = pid_x(transl_x_cmd)
            cf2.update_x(round(-cval_x, 1))
        time.sleep(0.03)
    except (UnboundLocalError, TypeError, ZeroDivisionError):
        print(err_dict["5"])
        print("Error: ", sys.exc_info()[0])
        cam.release()
        cv2.destroyAllWindows()
        cf2.land()
        traceback.print_exc()
        scf.close_link()

    # Moving avg filter ,battery:
```

```
                if cf2_bat is not None:
                    arb = np.append(arb, cf2_bat)
                    cf2_bat = np.average(arb)
                    if np.size(arb) >= samples_max:
                        arb = np.delete(arb, 0)

                # Do some logging of data for plotting
                # if marker_psi is not None or nf < nf_max:
                setpoint = np.append(setpoint, marker_psi)  # Commanded yaw
                yaw_cf = np.append(yaw_cf, cf2_psi)  # Actual yaw of CF
                pitch_cf = np.append(pitch_cf, cf2_theta)
                roll_cf = np.append(roll_cf, cf2_phi)
                ar_time = np.append(ar_time, dt)    # Time vector
                r = np.append(r, f.x[0, 0])  # Estimated yaw
                r2 = np.append(r2, f.x[2, 0])  # Estimated z distance
                r3 = np.append(r3, f.x[4, 0])   # Estimated x distance
                r4 = np.append(r4, f.x[6, 0])   # Estimated y distance
                ar_bat = np.append(ar_bat, cf2_bat)  # Battery level

                A = scf.is_link_open()  # Check that link is still open
            # ****** END OF MAIN LOOP ********* #
            if cf2_bat <= low_battery:
                print(err_dict["4"])
                print("Battery level: " + str(cf2_bat))
                cf2.land()
            else:
                print("Test completed. landing...")
                # Test sending a signal to CF: Move forward [cm]:
                # arguments: Forward(+), left(+), up(+)
                # if values are negative; opposite direction
                cf2.move_distance(round(transl_z_cmd-15), 0, 0)
                # Land the CF2
                time.sleep(0.1)
                cf2.land()
                print("Landed.")

# Important to close webcam connection when everything is done
if 'scf' in locals():
    scf.close_link()    # Close connection to CF
out.release()
cam.release()
cv2.destroyAllWindows()
if len(available) != 0:
    # Save data to yaml-file for analysis
    pid_settings = np.array([Kp_psi, Kd_psi, Ki_psi, turnRate])
    data = {'timestamp': np.asarray(ar_time).tolist(),
            'battery_level': np.asarray(ar_bat).tolist(),
            'yaw_cf': np.asarray(yaw_cf).tolist(),
            'roll_cf': np.asarray(roll_cf).tolist(),
            'pitch_cf': np.asarray(pitch_cf).tolist(),
            'yaw_commanded': np.asarray(setpoint).tolist(),
```

```
            'Kalman_yaw_estimation': np.asarray(r).tolist(),
            'Kalman_z_dist_estimation': np.asarray(r2).tolist(),
            'Kalman_x_dist_estimation': np.asarray(r3).tolist(),
            'Kalman_y_dist_estimation': np.asarray(r4).tolist(),
            'PID_settings': np.asarray(pid_settings).tolist()}

    # Save data to file
    with open(dumpfile_name, "w") as f:
            yaml.dump(data, f)

    # Plot results
    plotData(dumpfile_name, figname, figtitle)
```